

rxjava

Rx介绍

ReactiveX的历史

ReactiveX是Reactive Extensions的缩写，一般简写为Rx，最初是LINQ的一个扩展，由微软的架构师Erik Meijer领导的团队开发，在2012年11月开源，Rx是一个编程模型，目标是提供一致的编程接口，帮助开发者更方便的处理异步数据流，Rx库支持.NET、JavaScript和C++，Rx近几年越来越流行了，现在已经支持几乎全部的流行编程语言了，Rx的大部分语言库由ReactiveX这个组织负责维护，比较流行的有RxJava/RxJS/Rx.NET，社区网站是 reactivex.io。

什么是ReactiveX

微软给的定义是，Rx是一个函数库，让开发者可以利用可观察序列和LINQ风格查询操作符来编写异步和基于事件的程序，使用Rx，开发者可以用Observables表示异步数据流，用LINQ操作符查询异步数据流，用Schedulers参数化异步数据流的并发处理，Rx可以这样定义：Rx = Observables + LINQ + Schedulers。

ReactiveX.io给的定义是，Rx是一个使用可观察数据流进行异步编程的编程接口，ReactiveX结合了观察者模式、迭代器模式和函数式编程的精华。

ReactiveX的应用

很多公司都在使用ReactiveX，例如Microsoft、Netflix、Github、Trello、SoundCloud。

ReactiveX宣言

ReactiveX不仅仅是一个编程接口，它是一种编程思想的突破，它影响了许多其它的程序库和框架以及编程语言。

Rx模式

使用观察者模式

- 创建： Rx可以方便的创建事件流和数据流
- 组合： Rx使用查询式的操作符组合和变换数据流
- 监听： Rx可以订阅任何可观察的数据流并执行操作

简化代码

- **函数式风格**: 对可观察数据流使用无副作用的输入输出函数，避免了程序里错综复杂的状态
- **简化代码**: Rx的操作符通常可以将复杂的难题简化为很少的几行代码
- **异步错误处理**: 传统的try/catch没办法处理异步计算，Rx提供了合适的错误处理机制
- **轻松使用并发**: Rx的Observables和Schedulers让开发者可以摆脱底层的线程同步和各种并发问题

使用Observable的优势

Rx扩展了观察者模式用于支持数据和事件序列，添加了一些操作符，它让你可以声明式的组合这些序列，而无需关注底层的实现：如线程、同步、线程安全、并发数据结构和非阻塞IO。

Observable通过使用最佳的方式访问异步数据序列填补了这个间隙

	单个数据	多个数据
同步	<code>T getData()</code>	<code>Iterable<T> getData()</code>
异步	<code>Future<T> getData()</code>	<code>Observable<T> getData()</code>

Rx的Observable模型让你可以像使用集合数据一样操作异步事件流，对异步事件流使用各种简单、可组合的操作。

Observable可组合

对于单层的异步操作来说，Java中Future对象的处理方式是非常简单有效的，但是一旦涉及到嵌套，它们就开始变得异常繁琐和复杂。使用Future很难很好的组合带条件的异步执行流程（考虑到运行时各种潜在的问题，甚至可以说是不可能的），当然，要想实现还是可以做到的，但是非常困难，或许你可以用`Future.get()`，但这样做，异步执行的优势就完全没有了。从另一方面说，Rx的Observable一开始就是为组合异步数据流准备的。

Observable更灵活

Rx的Observable不仅支持处理单独的标量值（就像Future可以做的），也支持数据序列，甚至是无穷的数据流。`Observable`是一个抽象概念，适用于任何场景。Observable拥有它的近亲Iterable的全部优雅与灵活。

Observable是异步的双向push，Iterable是同步的单向pull，对比：

事件	ITERABLE(PULL)	OBSERVABLE(PUSH)
获取数据	<code>T next()</code>	<code>onNext(T)</code>
异常处理	<code>throws Exception</code>	<code>onError(Exception)</code>
任务完成	<code>!hasNext()</code>	<code>onCompleted()</code>

Observable无偏见

Rx对于对于并发性或异步性没有任何特殊的偏好，**Observable**可以用任何方式实现，线程池、事件循环、非阻塞IO、Actor模式，任何满足你的需求的，你擅
长或偏好的方式都可以。无论你选择怎样实现它，无论底层实现是阻塞的还是非
阻塞的，客户端代码将所有与**Observable**的交互都当做是异步的。

Observable是如何实现的？

```
public Observable<data> getData();
```

- 它能与调用者在同一线程同步执行吗？
- 它能异步地在单独的线程执行吗？
- 它会将工作分发到多个线程，返回数据的顺序是任意的吗？
- 它使用Actor模式而不是线程池吗？
- 它使用NIO和事件循环执行异步网络访问吗？
- 它使用事件循环将工作线程从回调线程分离出来吗？

从**Observer**的视角看，这些都无所谓，重要的是：使用Rx，你可以改变你的观念，你可以在完全不影响**Observable**程序库使用者的情况下，彻底的改变
Observable的底层实现。

使用回调存在很多问题

回调在不阻塞任何事情的情况下，解决了 `Future.get()` 过早阻塞的问题。由于响应结果一旦就绪Callback就会被调用，它们天生就是高效率的。不过，就像使用Future一样，对于单层的异步执行来说，回调很容易使用，对于嵌套的异步组合，它们显得非常笨拙。

Rx是一个多语言的实现

Rx在大量的编程语言中都有实现，并尊重实现语言的风格，而且更多的实现正在飞速增加。

响应式编程

Rx提供了一系列的操作符，你可以使用它们来过滤(filter)、选择(select)、变换(transform)、结合(combine)和组合(compose)多个**Observable**，这些操作符让执行和复合变得非常高效。

你可以把**Observable**当做Iterable的推送方式的等价物，使用Iterable，消费者从生产者那拉取数据，线程阻塞直至数据准备好。使用**Observable**，在数据准备好时，生产者将数据推送给消费者。数据可以同步或异步的到达，这种方式更灵活。

下面的例子展示了相似的高阶函数在Iterable和Observable上的应用

```

// Iterable
getDataFromLocalMemory()
  .skip(10)
  .take(5)
  .map({ s -> return s + " transformed" })
  .foreach({ println "next => " + it })

// Observable
getDataFromNetwork()
  .skip(10)
  .take(5)
  .map({ s -> return s + " transformed" })
  .subscribe({ println "onNext => " + it })

```

Observable类型给GOF的观察者模式添加了两种缺少的语义，这样就和Iterable类型中可用的操作一致了：

1. 生产者可以发信号给消费者，通知它没有更多数据可用了（对于Iterable，一个for循环正常完成表示没有数据了；对于Observable，就是调用观察者的`onCompleted`方法）
2. 生产者可以发信号给消费者，通知它遇到了一个错误（对于Iterable，迭代过程中发生错误会抛出异常；对于Observable，就是调用观察者（Observer）的`onError`方法）

有了这两种功能，Rx就能使Observable与Iterable保持一致了，唯一的不同是数据流的方向。任何对Iterable的操作，你都可以对Observable使用。

名词定义

这里给出一些名词的翻译

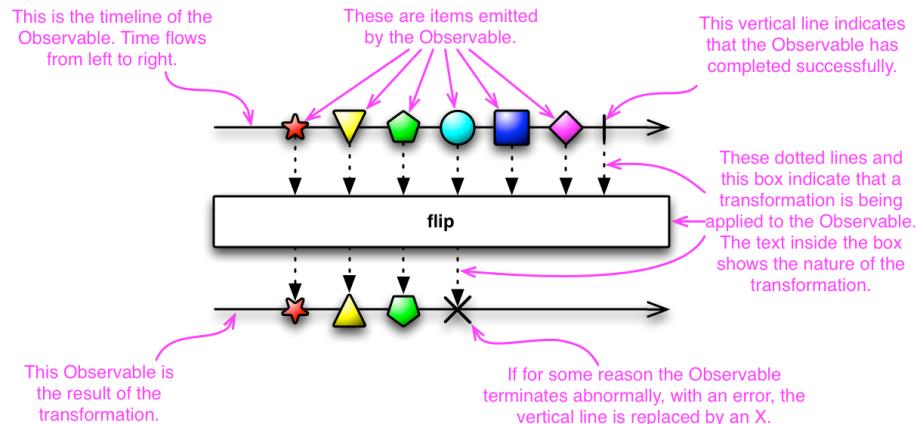
- Reactive 直译为反应性的，有活性的，根据上下文一般翻译为反应式、响应式
- Iterable 可迭代对象，支持以迭代器的形式遍历，许多语言中都存在这个概念
- Observable 可观察对象，在Rx中定义为更强大的Iterable，在观察者模式中是被观察的对象，一旦数据产生或发生变化，会通过某种方式通知观察者或订阅者
- Observer 观察者对象，监听Observable发射的数据并做出响应，Subscriber是它的一个特殊实现
- emit 直译为发射，发布，发出，含义是Observable在数据产生或变化时发送通知给Observer，调用Observer对应的方法，文章里一律译为发射
- items 直译为项目，条目，在Rx里是指Observable发射的数据项，文章里一律译为数据，数据项

Observable

概述

在ReactiveX中，一个观察者(Observer)订阅一个可观察对象(Observable)。观察者对Observable发射的数据或数据序列作出响应。这种模式可以极大地简化并发操作，因为它创建了一个处于待命状态的观察者哨兵，在未来某个时刻响应Observable的通知，不需要阻塞等待Observable发射数据。

这篇文章会解释什么是响应式编程模式(reactive pattern)，以及什么是可观察对象(Observables)和观察者(observers)，其它几篇文章会展示如何用操作符组合和改变Observable的行为。



相关参考：

- [Single](#) - 一个特殊的Observable，只发射单个数据。

背景知识

在很多软件编程任务中，或多或少你都会期望你写的代码能按照编写的顺序，一次一个的顺序执行和完成。但是在ReactiveX中，很多指令可能是并行执行的，之后他们的执行结果才会被观察者捕获，顺序是不确定的。为达到这个目的，你定义一种获取和变换数据的机制，而不是调用一个方法。在这种机制下，存在一个可观察对象(Observable)，观察者(Observer)订阅(Subscribe)它，当数据就绪时，之前定义的机制就会分发数据给一直处于等待状态的观察者哨兵。

这种方法的优点是，如果你有大量的任务要处理，它们互相之间没有依赖关系。你可以同时开始执行它们，不用等待一个完成再开始下一个（用这种方式，你的整个任务队列能耗费的最长时间，不会超过任务里最耗时的那个）。

有很多术语可用于描述这种异步编程和设计模式，在本文里我们使用这些术语：一个观察者订阅一个可观察对象 (*An observer subscribes to an Observable*)。通过调用观察者的方法，Observable发射数据或通知给它的观察者。

在其它的文档和场景里，有时我们也将**Observer**叫做**Subscriber**、**Watcher**、**Reactor**。这个模型通常被称作**Reactor**模式。

创建观察者

本文使用类似于Groovy的伪代码举例，但是ReactiveX有多种语言的实现。

普通的方法调用（不是某种异步方法，也不是Rx中的并行调用），流程通常是指这样的：

1. 调用某一个方法
2. 用一个变量保存方法返回的结果
3. 使用这个变量和它的新值做些有用的事

用代码描述就是：

```
// make the call, assign its return value to `returnValue`  
returnValue = someMethod(itsParameters);  
// do something useful with returnValue
```

在异步模型中流程更像这样的：

1. 定义一个方法，这个方法拿着某个异步调用的返回值做一些有用的事情。这个方法是观察者的一部分。
2. 将这个异步调用本身定义为一个Observable
3. 观察者通过订阅(Subscribe)操作关联到那个Observable
4. 继续你的业务逻辑，等方法返回时，Observable会发射结果，观察者的方法会开始处理结果或结果集

用代码描述就是：

```
// defines, but does not invoke, the subscriber's onNext  
handler  
// (in this example, the observer is very simple and has  
only an onNext handler)  
def myOnNext = { it -> do something useful with it };  
// defines, but does not invoke, the observable  
def myObservable = someObservable(itsParameters);  
// subscribes the subscriber to the observable, and  
invokes the observable  
myObservable.subscribe(myOnNext);  
// go on about my business
```

回调方法 (onNext, onCompleted, onError)

Subscribe方法用于将观察者连接到Observable，你的观察者需要实现以下方法的一个子集：

- **onNext(T item)**

Observable调用这个方法发射数据，方法的参数就是Observable发射的数据，这个方法可能会被调用多次，取决于你的实现。

- **onError(Exception ex)**

当Observable遇到错误或者无法返回期望的数据时会调用这个方法，这个调用会终止Observable，后续不会再调用onNext和onCompleted，onError方法的参数是抛出的异常。

- **onComplete**

正常终止，如果没有遇到错误，**Observable**在最后一次调用onNext之后调用此方法。

根据**Observable**协议的定义，onNext可能会被调用零次或者很多次，最后会有一次onCompleted或onError调用（不会同时），传递数据给onNext通常被称作发射，onCompleted和onError被称作通知。

下面是一个更完整的例子：

```
def myOnNext      = { item -> /* do something useful with
item */ };
def myError       = { throwable -> /* react sensibly to a
failed call */ };
def myComplete    = { /* clean up after the final response
*/ };
def myObservable = someMethod(itsParameters);
myObservable.subscribe(myOnNext, myError, myComplete);
// go on about my business
```

取消订阅 (Unsubscribing)

在一些ReactiveX实现中，有一个特殊的观察者接口*Subscriber*，它有一个*unsubscribe*方法。调用这个方法表示你不关心当前订阅的**Observable**了，因此**Observable**可以选择停止发射新的数据项（如果没有其它观察者订阅）。

取消订阅的结果会传递给这个**Observable**的操作符链，而且会导致这个链条上的每个环节都停止发射数据项。这些并不保证会立即发生，然而，对一个**Observable**来说，即使没有观察者了，它也可以在一个while循环中继续生成并尝试发射数据项。

关于命名约定

ReactiveX的每种特定语言的实现都有自己的命名偏好，虽然不同的实现之间有很多共同点，但并不存在一个统一的命名标准。

而且，在某些场景中，一些名字有不同的隐含意义，或者在某些语言看来比较怪异。

例如，有一个*onEvent*命名模式(onNext, onCompleted, onError)，在一些场景中，这些名字可能意味着事件处理器已经注册。然而在ReactiveX里，他们是事件处理器的名字。

Observables的"热"和"冷"

Observable什么时候开始发射数据序列？这取决于**Observable**的实现，一个"热"的**Observable**可能一创建完就开始发射数据，因此所有后续订阅它的观察者可能从序列中间的某个位置开始接受数据（有一些数据错过了）。一个"冷"的**Observable**会一直等待，直到有观察者订阅它才开始发射数据，因此这个观察者可以确保会收到整个数据序列。

在一些ReactiveX实现里，还存在一种被称作`Connectable`的Observable，不管有没有观察者订阅它，这种Observable都不会开始发射数据，除非`Connect`方法被调用。

用操作符组合Observable

对于ReactiveX来说，Observable和Observer仅仅是个开始，它们本身不过是标准观察者模式的一些轻量级扩展，目的是为了更好的处理事件序列。

ReactiveX真正强大的地方在于它的操作符，操作符让你可以变换、组合、操纵和处理Observable发射的数据。

Rx的操作符让你可以用声明式的风格组合异步操作序列，它拥有回调的所有效率优势，同时又避免了典型的异步系统中嵌套回调的缺点。

下面是常用的操作符列表：

1. **创建操作** Create, Defer, Empty/Never/Throw, From, Interval, Just, Range, Repeat, Start, Timer
2. **变换操作** Buffer, FlatMap, GroupBy, Map, Scan和Window
3. **过滤操作** Debounce, Distinct, ElementAt, Filter, First, IgnoreElements, Last, Sample, Skip, SkipLast, Take, TakeLast
4. **组合操作** And/Then/When, CombineLatest, Join, Merge, StartWith, Switch, Zip
5. **错误处理** Catch和Retry
6. **辅助操作** Delay, Do, Materialize/Dematerialize, ObserveOn, Serialize, Subscribe, SubscribeOn, TimeInterval, Timeout, Timestamp, Using
7. **条件和布尔操作** All, Amb, Contains, DefaultIfEmpty, SequenceEqual, SkipUntil, SkipWhile, TakeUntil, TakeWhile
8. **算术和集合操作** Average, Concat, Count, Max, Min, Reduce, Sum
9. **转换操作** To
10. **连接操作** Connect, Publish, RefCount, Replay
11. **反压操作**，用于增加特殊的流程控制策略的操作符

这些操作符并不全都是ReactiveX的核心组成部分，有一些是语言特定的实现或可选的模块。

RxJava

在RxJava中，一个实现了`Observer`接口的对象可以订阅(`subscribe`)一个`Observable`类的实例。订阅者(subscriber)对Observable发射(emit)的任何数据或数据序列作出响应。这种模式简化了并发操作，因为它不需要阻塞等待Observable发射数据，而是创建了一个处于待命状态的观察者哨兵，哨兵在未来某个时刻响应Observable的通知。

Single

RxJava（以及它派生出来的RxGroovy和RxScala）中有一个名为**Single**的Observable变种。

Single类似于Observable，不同的是，它总是只发射一个值，或者一个错误通知，而不是发射一系列的值。

因此，不同于Observable需要三个方法onNext, onError, onCompleted，订阅Single只需要两个方法：

- **onSuccess** - Single发射单个的值到这个方法
- **onError** - 如果无法发射需要的值，Single发射一个Throwable对象到这个方法

Single只会调用这两个方法中的一个，而且只会调用一次，调用了任何一个方法之后，订阅关系终止。

Single的操作符

Single也可以组合使用多种操作，一些操作符让你可以混合使用Observable和Single：

操作符	返回值	说明
compose	Single	创建一个自定义的操作符
concat and concatWith	Observable	连接多个Single和Observable发射的数据
create	Single	调用观察者的create方法创建一个Single
error	Single	返回一个立即给订阅者发射错误通知的Single
flatMap	Single	返回一个Single，它发射对原Single的数据执行flatMap操作后的结果
flatMapObservable	Observable	返回一个Observable，它发射对原Single的数据执行flatMap操作后的结果
from	Single	将Future转换成Single
just	Single	返回一个发射一个指定值的Single
map	Single	返回一个Single，它发射对原Single的数据执行map操作后的结果
merge	Single	将一个Single(它发射的数据是另一个Single，假设为B)转换成另一个Single(它发射来自另一个Single(B)的数据)
merge and mergeWith	Observable	合并发射来自多个Single的数据
observeOn	Single	指示Single在指定的调度程序上调用订阅者的方法
onErrorReturn	Single	将一个发射错误通知的Single转换成一个发射指定数据项的Single
subscribeOn	Single	指示Single在指定的调度程序上执行操作
timeout	Single	它给原有的Single添加超时控制，如果超时了就发射一个错误通知

操作符	返回值	说明
toSingle	Single	将一个发射单个值的Observable转换为一个Single
zip and zipWith	Single	将多个Single转换为一个，后者发射的数据是对前者应用一个函数后的结果

操作符图示

详细的图解可以参考英文文档: [Single](#)

Subject

Subject可以看成是一个桥梁或者代理，在某些ReactiveX实现中（如RxJava），它同时充当了Observer和Observable的角色。因为它是一个Observer，它可以订阅一个或多个Observable；又因为它是一个Observable，它可以转发它收到(Observe)的数据，也可以发射新的数据。

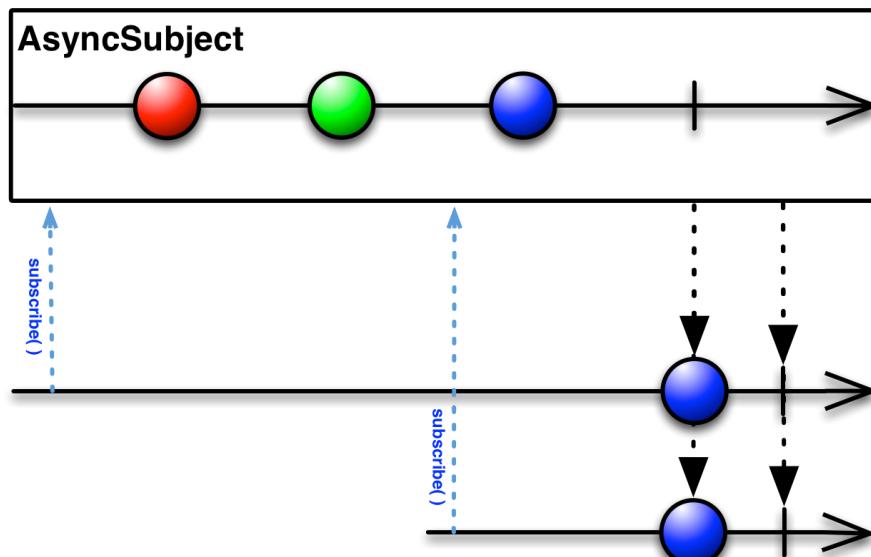
由于一个Subject订阅一个Observable，它可以触发这个Observable开始发射数据（如果那个Observable是“冷”的--就是说，它等待有订阅才开始发射数据）。因此有这样的效果，Subject可以把原来那个“冷”的Observable变成“热”的。

Subject的种类

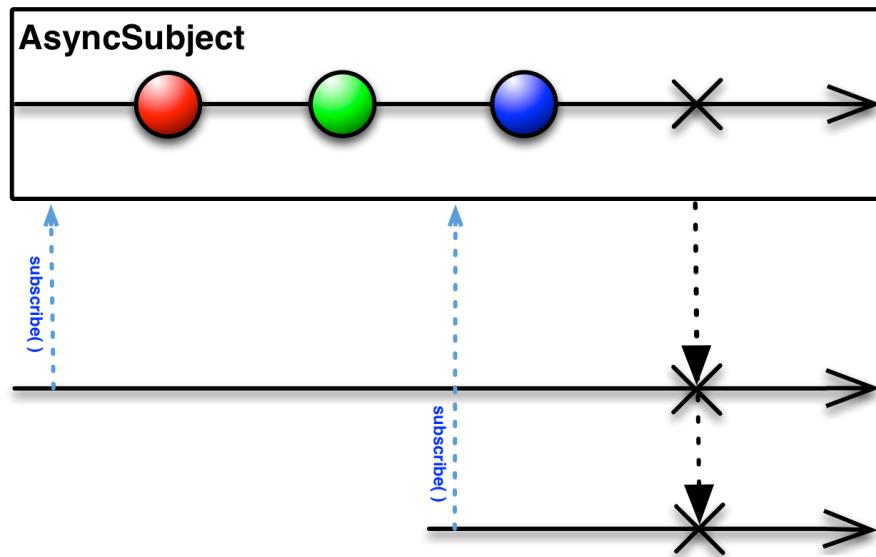
针对不同的场景一共有四种类型的Subject。他们并不是在所有的实现中全部都存在，而且一些实现使用其它的命名约定（例如，在RxScala中Subject被称作PublishSubject）。

AsyncSubject

一个AsyncSubject只在原始Observable完成后，发射来自原始Observable的最后一个值。（如果原始Observable没有发射任何值，AsyncObject也不发射任何值）它会把这最后一个值发射给任何后续的观察者。

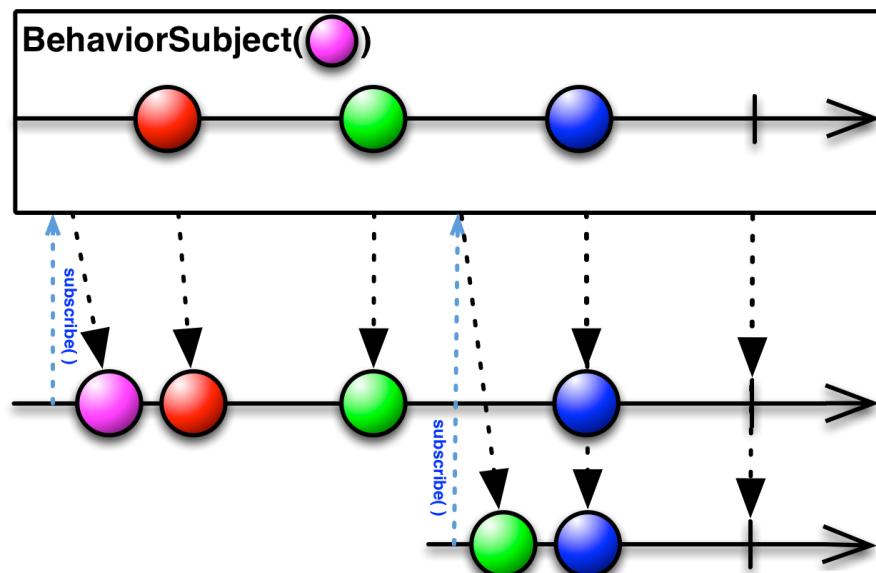


然而，如果原始的Observable因为发生了错误而终止，AsyncSubject将不会发射任何数据，只是简单的向前传递这个错误通知。

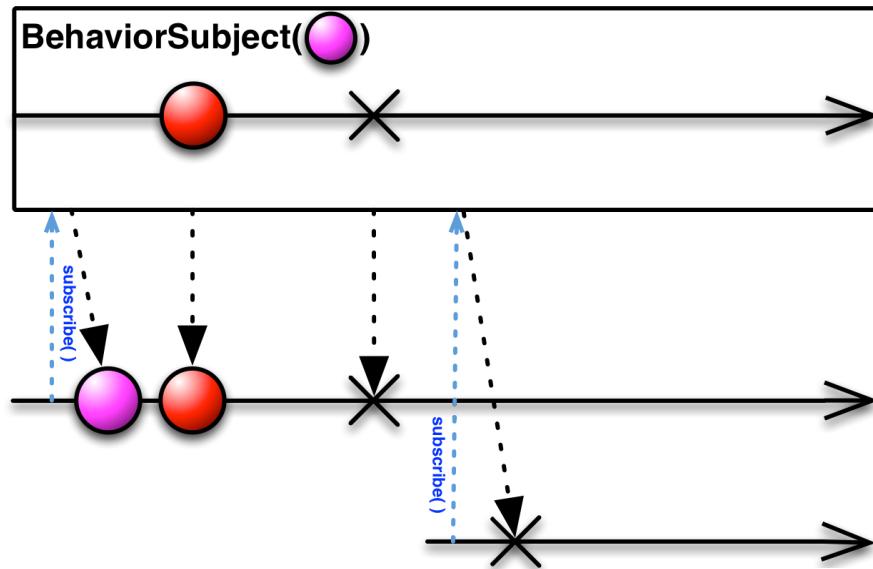


BehaviorSubject

当观察者订阅BehaviorSubject时，它开始发射原始Observable最近发射的数据（如果此时还没有收到任何数据，它会发射一个默认值），然后继续发射其它任何来自原始Observable的数据。

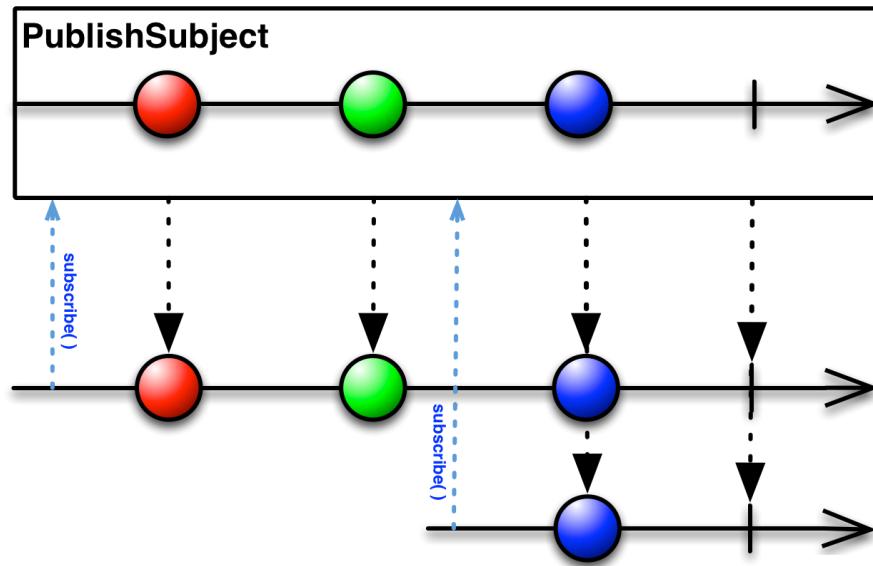


然而，如果原始的Observable因为发生了一个错误而终止，BehaviorSubject将不会发射任何数据，只是简单的向前传递这个错误通知。

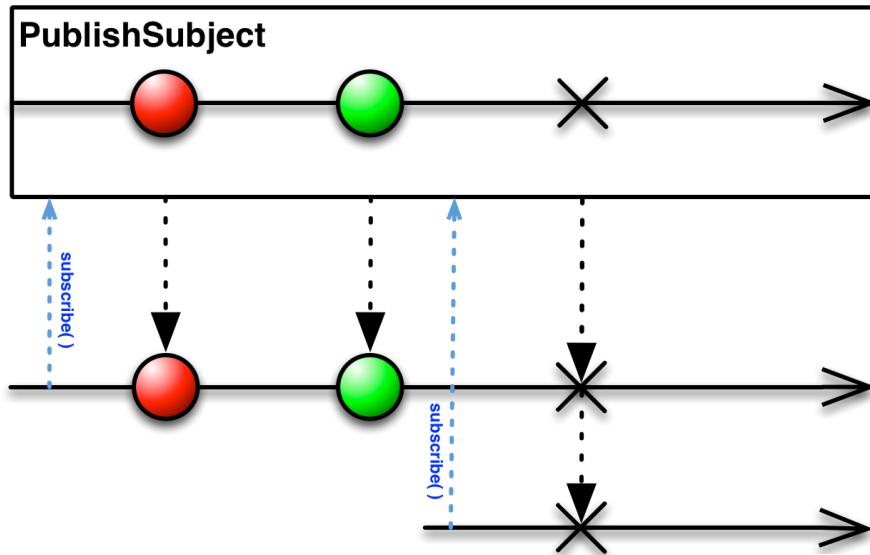


PublishSubject

PublishSubject只会把在订阅发生的时间点之后来自原始Observable的数据发射给观察者。需要注意的是，PublishSubject可能会一创建完成就立刻开始发射数据（除非你可以阻止它发生），因此这里有一个风险：在Subject被创建后到有观察者订阅它之前这个时间段内，一个或多个数据可能会丢失。如果要确保来自原始Observable的所有数据都被分发，你需要这样做：或者使用Create创建那个Observable以便手动给它引入"冷"Observable的行为（当所有观察者都已经订阅时才开始发射数据），或者改用ReplaySubject。



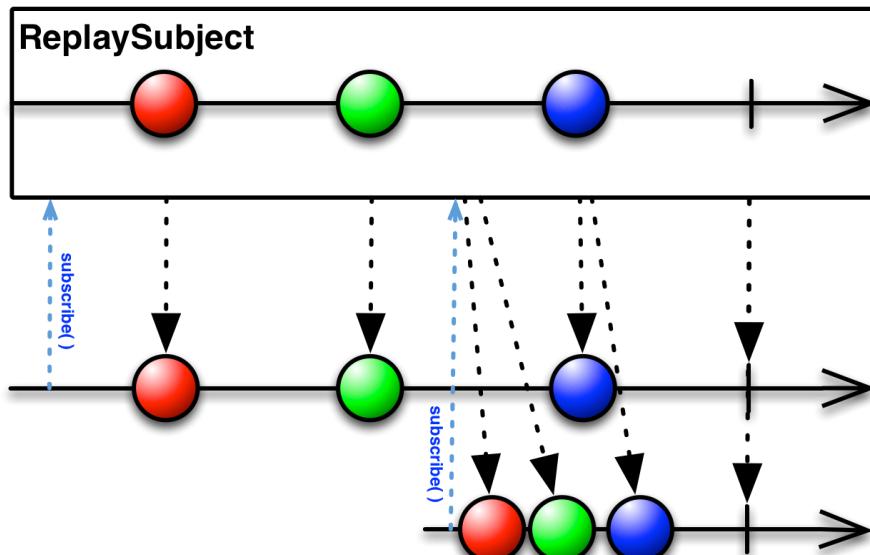
如果原始的Observable因为发生了一个错误而终止，PublishSubject将不会发射任何数据，只是简单的向前传递这个错误通知。



ReplaySubject

ReplaySubject会发射所有来自原始Observable的数据给观察者，无论它们是什么时候订阅的。也有其它版本的ReplaySubject，在重放缓存增长到一定大小的时候或过了一段时间后会丢弃旧的数据（原始Observable发射的）。

如果你把ReplaySubject当作一个观察者使用，注意不要从多个线程中调用它的onNext方法（包括其它的on系列方法），这可能导致同时（非顺序）调用，这会违反Observable协议，给Subject的结果增加了不确定性。



RxJava的对应类

假设你有一个Subject，你想把它传递给其它的代理或者暴露它的Subscriber接口，你可以调用它的asObservable方法，这个方法返回一个Observable。具体使用方法可以参考Javadoc文档。

串行化

如果你把 `Subject` 当作一个 `Subscriber` 使用，注意不要从多个线程中调用它的 `onNext` 方法（包括其它的 `on` 系列方法），这可能导致同时（非顺序）调用，这会违反 `Observable` 协议，给 `Subject` 的结果增加了不确定性。

要避免此类问题，你可以将 `Subject` 转换为一个 `SerializedSubject`，类似于这样：

```
mySafeSubject = new SerializedSubject( myUnsafeSubject );
```

调度器 Scheduler

如果你想给 `Observable` 操作符链添加多线程功能，你可以指定操作符（或者特定的 `Observable`）在特定的调度器 (`Scheduler`) 上执行。

某些 `ReactiveX` 的 `Observable` 操作符有一些变体，它们可以接受一个 `Scheduler` 参数。这个参数指定操作符将它们的部分或全部任务放在一个特定的调度器上执行。

使用 `ObserveOn` 和 `SubscribeOn` 操作符，你可以让 `Observable` 在一个特定的调度器上执行，`ObserveOn` 指示一个 `Observable` 在一个特定的调度器上调用观察者的 `onNext`, `onError` 和 `onCompleted` 方法，`SubscribeOn` 更进一步，它指示 `Observable` 将全部的处理过程（包括发射数据和通知）放在特定的调度器上执行。

RxJava示例

调度器的种类

下表展示了 RxJava 中可用的调度器种类：

调度器类型	效果
<code>Schedulers.computation()</code>	用于计算任务，如事件循环或回调处理，不要用于 IO 操作 (IO 操作请使用 <code>Schedulers.io()</code>)；默认线程数等于处理器的数量
<code>Schedulers.from(executor)</code>	使用指定的 <code>Executor</code> 作为调度器
<code>Schedulers.immediate()</code>	在当前线程立即开始执行任务
<code>Schedulers.io()</code>	用于 IO 密集型任务，如异步阻塞 IO 操作，这个调度器的线程池会根据需要增长；对于普通的计算任务，请使用 <code>Schedulers.computation()</code> ； <code>Schedulers.io()</code> 默认是一个 <code>CachedThreadScheduler</code> ，很像一个有线程缓存的新线程调度器
<code>Schedulers.newThread()</code>	为每个任务创建一个新线程
<code>Schedulers.trampoline()</code>	当其它排队的任务完成后，在当前线程排队开始执行

默认调度器

在RxJava中，某些Observable操作符的变体允许你设置用于操作执行的调度器，其它的则不在任何特定的调度器上执行，或者在一个指定的默认调度器上执行。下面的表格列出来了一些操作符的默认调度器：

操作符	调度器
buffer(timespan)	computation
buffer(timespan, count)	computation
buffer(timespan, timeshift)	computation
debounce(timeout, unit)	computation
delay(delay, unit)	computation
delaySubscription(delay, unit)	computation
interval	computation
repeat	trampoline
replay(time, unit)	computation
replay(bufferSize, time, unit)	computation
replay(selector, time, unit)	computation
replay(selector, bufferSize, time, unit)	computation
retry	trampoline
sample(period, unit)	computation
skip(time, unit)	computation
skipLast(time, unit)	computation
take(time, unit)	computation
takeLast(time, unit)	computation
takeLast(count, time, unit)	computation
takeLastBuffer(time, unit)	computation
takeLastBuffer(count, time, unit)	computation
throttleFirst	computation
throttleLast	computation
throttleWithTimeout	computation
timeInterval	immediate
timeout(timeoutSelector)	immediate
timeout(firstTimeoutSelector, timeoutSelector)	immediate
timeout(timeoutSelector, other)	immediate
timeout(timeout, timeUnit)	computation
timeout(firstTimeoutSelector, timeoutSelector, other)	immediate
timeout(timeout, timeUnit, other)	computation
timer	computation
timestamp	immediate
window(timespan)	computation
window(timespan, count)	computation

使用调度器

除了将这些调度器传递给RxJava的Observable操作符，你也可以用它们调度你自己的任务。下面的示例展示了Scheduler.Worker的用法：

```
worker = Schedulers.newThread().createWorker();
worker.schedule(new Action0() {

    @Override
    public void call() {
        yourwork();
    }

});
// some time later...
worker.unsubscribe();
```

递归调度器

要调度递归的方法调用，你可以使用schedule，然后再用schedule(this)，示例：

```
worker = Schedulers.newThread().createWorker();
worker.schedule(new Action0() {

    @Override
    public void call() {
        yourwork();
        // recurse until unsubscribed (schedule will do
        // nothing if unsubscribed)
        worker.schedule(this);
    }

});
// some time later...
worker.unsubscribe();
```

检查或设置取消订阅状态

Worker类的对象实现了Subscription接口，使用它的isUnsubscribed和unsubscribe方法，所以你可以在订阅取消时停止任务，或者从正在调度的任务内部取消订阅，示例：

```
worker worker = Schedulers.newThread().createWorker();
Subscription mySubscription = worker.schedule(new
Action0() {

    @Override
    public void call() {
        while(!worker.isUnsubscribed()) {
            status = yourWork();
            if(QUIT == status) { worker.unsubscribe(); }
        }
    }
});
```

Worker同时是Subscription，因此你可以（通常也应该）调用它的unsubscribe方法通知可以挂起任务和释放资源了。

延时和周期调度器

你可以使用schedule(action,delayTime,timeUnit)在指定的调度器上延时执行你的任务，下面例子中的任务将在500毫秒之后开始执行：

```
someScheduler.schedule(someAction, 500,
TimeUnit.MILLISECONDS);
```

使用另一个版本的schedule，
schedulePeriodically(action,initialDelay,period,timeUnit)方法让你可以安排一个定期执行的任务，下面例子的任务将在500毫秒之后执行，然后每250毫秒执行一次：

```
someScheduler.schedulePeriodically(someAction, 500, 250,
TimeUnit.MILLISECONDS);
```

测试调度器

TestScheduler让你可以对调度器的时钟表现进行手动微调。这对依赖精确时间安排的任务的测试很有用处。这个调度器有三个额外的方法：

- advanceTimeTo(time,unit) 向前波动调度器的时钟到一个指定的时间点
- advanceTimeBy(time,unit) 将调度器的时钟向前拨动一个指定的时间段
- triggerActions() 开始执行任何计划中的但是未启动的任务，如果它们的计划时间等于或者早于调度器时钟的当前时间

操作符分类

ReactiveX的每种编程语言的实现都实现了一组操作符的集合。不同的实现之间有很多重叠的部分，也有一些操作符只存在特定的实现中。每种实现都倾向于用那种编程语言中他们熟悉的上下文中相似的方法给这些操作符命名。

本文首先会给出ReactiveX的核心操作符列表和对应的文档链接，后面还有一个决策树用于帮助你根据具体的场景选择合适的操作符。最后有一个语言特定实现的按字母排序的操作符列表。

如果你想实现你自己的操作符，可以参考这里：[实现自定义操作符](#)

创建操作

用于创建Observable的操作符

- [Create](#) — 通过调用观察者的方法从头创建一个Observable
- [Defer](#) — 在观察者订阅之前不创建这个Observable，为每一个观察者创建一个新的Observable
- [Empty/Never/Throw](#) — 创建行为受限的特殊Observable
- [From](#) — 将其它的对象或数据结构转换为Observable
- [Interval](#) — 创建一个定时发射整数序列的Observable
- [Just](#) — 将对象或者对象集合转换为一个会发射这些对象的Observable
- [Range](#) — 创建发射指定范围的整数序列的Observable
- [Repeat](#) — 创建重复发射特定的数据或数据序列的Observable
- [Start](#) — 创建发射一个函数的返回值的Observable
- [Timer](#) — 创建在一个指定的延迟之后发射单个数据的Observable

变换操作

这些操作符可用于对Observable发射的数据进行变换，详细解释可以看每个操作符的文档

- [Buffer](#) — 缓存，可以简单的理解为缓存，它定期从Observable收集数据到一个集合，然后把这些数据集合打包发射，而不是一次发射一个
- [FlatMap](#) — 扁平映射，将Observable发射的数据变换为Observables集合，然后将这些Observable发射的数据平坦化的放进一个单独的Observable，可以认为是一个将嵌套的数据结构展开的过程。
- [GroupBy](#) — 分组，将原来的Observable分拆为Observable集合，将原始Observable发射的数据按Key分组，每一个Observable发射一组不同的数据
- [Map](#) — 映射，通过对序列的每一项都应用一个函数变换Observable发射的数据，实质是对序列中的每一项执行一个函数，函数的参数就是这个数据项
- [Scan](#) — 扫描，对Observable发射的每一项数据应用一个函数，然后按顺序依次发射这些值
- [Window](#) — 窗口，定期将来自Observable的数据分拆成一些Observable窗口，然后发射这些窗口，而不是每次发射一项。类似于Buffer，但Buffer发射的是数据，Window发射的是Observable，每一个Observable发射原始Observable的数据的一个子集

过滤操作

这些操作符用于从Observable发射的数据中进行选择

- `Debounce` — 只有在空闲了一段时间后才发射数据，通俗的说，就是如果一段时间没有操作，就执行一次操作
- `Distinct` — 去重，过滤掉重复数据项
- `ElementAt` — 取值，取特定位置的数据项
- `Filter` — 过滤，过滤掉没有通过谓词测试的数据项，只发射通过测试的
- `First` — 首项，只发射满足条件的第一条数据
- `IgnoreElements` — 忽略所有的数据，只保留终止通知(onError或onCompleted)
- `Last` — 末项，只发射最后一条数据
- `Sample` — 取样，定期发射最新的数据，等于是数据抽样，有的实现里叫ThrottleFirst
- `Skip` — 跳过前面的若干项数据
- `SkipLast` — 跳过后面的若干项数据
- `Take` — 只保留前面的若干项数据
- `TakeLast` — 只保留后面的若干项数据

组合操作

组合操作符用于将多个Observable组合成一个单一的Observable

- `And/Then/when` — 通过模式(And条件)和计划(Then次序)组合两个或多个Observable发射的数据集
- `CombineLatest` — 当两个Observables中的任何一个发射了一个数据时，通过一个指定的函数组合每个Observable发射的最新数据（一共有两个数据），然后发射这个函数的结果
- `Join` — 无论何时，如果一个Observable发射了一个数据项，只要在另一个Observable发射的数据项定义的时间窗口内，就将两个Observable发射的数据合并发射
- `Merge` — 将两个Observable发射的数据组合并成一个
- `Startwith` — 在发射原来的Observable的数据序列之前，先发射一个指定的数据序列或数据项
- `Switch` — 将一个发射Observable序列的Observable转换为这样一个Observable：它逐个发射那些Observable最近发射的数据
- `Zip` — 打包，使用一个指定的函数将多个Observable发射的数据组合在一起，然后将这个函数的结果作为单项数据发射

错误处理

这些操作符用于从错误通知中恢复

- `Catch` — 捕获，继续序列操作，将错误替换为正常的数据，从onError通知中恢复
- `Retry` — 重试，如果Observable发射了一个错误通知，重新订阅它，期待它正常终止

辅助操作

一组用于处理Observable的操作符

- `Delay` — 延迟一段时间发射结果数据
- `Do` — 注册一个动作占用一些Observable的生命周期事件，相当于Mock某个操作
- `Materialize/Dematerialize` — 将发射的数据和通知都当做数据发射，或者反过来
- `ObserveOn` — 指定观察者观察Observable的调度程序（工作线程）
- `Serialize` — 强制Observable按次序发射数据并且功能是有效的
- `Subscribe` — 收到Observable发射的数据和通知后执行的操作
- `SubscribeOn` — 指定Observable应该在哪个调度程序上执行
- `TimeInterval` — 将一个Observable转换为发射两个数据之间所耗费时间的Observable
- `Timeout` — 添加超时机制，如果过了指定的一段时间没有发射数据，就发射一个错误通知
- `Timestamp` — 给Observable发射的每个数据项添加一个时间戳
- `Using` — 创建一个只在Observable的生命周期内存在的一次性资源

条件和布尔操作

这些操作符可用于单个或多个数据项，也可用于Observable

- `All` — 判断Observable发射的所有数据项是否都满足某个条件
- `Amb` — 给定多个Observable，只让第一个发射数据的Observable发射全部数据
- `Contains` — 判断Observable是否会发射一个指定的数据项
- `DefaultIfEmpty` — 发射来自原始Observable的数据，如果原始Observable没有发射数据，就发射一个默认数据
- `SequenceEqual` — 判断两个Observable是否按相同的数据序列
- `SkipUntil` — 丢弃原始Observable发射的数据，直到第二个Observable发射了一个数据，然后发射原始Observable的剩余数据
- `SkipWhile` — 丢弃原始Observable发射的数据，直到一个特定的条件为假，然后发射原始Observable剩余的数据
- `TakeUntil` — 发射来自原始Observable的数据，直到第二个Observable发射了一个数据或一个通知
- `TakeWhile` — 发射原始Observable的数据，直到一个特定的条件为真，然后跳过剩余的数据

算术和聚合操作

这些操作符可用于整个数据序列

- `Average` — 计算Observable发射的数据序列的平均值，然后发射这个结果
- `Concat` — 不交错的连接多个Observable的数据
- `Count` — 计算Observable发射的数据个数，然后发射这个结果
- `Max` — 计算并发射数据序列的最大值
- `Min` — 计算并发射数据序列的最小值
- `Reduce` — 按顺序对数据序列的每一个应用某个函数，然后返回这个值
- `Sum` — 计算并发射数据序列的和

连接操作

一些有精确可控的订阅行为的特殊Observable

- `Connect` — 指示一个可连接的Observable开始发射数据给订阅者
- `Publish` — 将一个普通的Observable转换为可连接的
- `RefCount` — 使一个可连接的Observable表现得像一个普通的Observable
- `Replay` — 确保所有的观察者收到同样的数据序列，即使他们在Observable开始发射数据之后才订阅

转换操作

- `To` — 将Observable转换为其它的对象或数据结构
- `blocking` 阻塞Observable的操作符

操作符决策树

几种主要的需求

- 直接创建一个Observable（创建操作）
- 组合多个Observable（组合操作）
- 对Observable发射的数据执行变换操作（变换操作）
- 从Observable发射的数据中取特定的值（过滤操作）
- 转发Observable的部分值（条件/布尔/过滤操作）
- 对Observable发射的数据序列求值（算术/聚合操作）

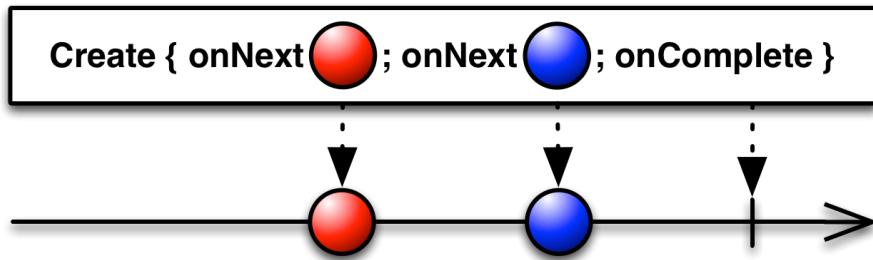
这个页面展示了创建Observable的各种方法。

- `just()` — 将一个或多个对象转换成发射这个或这些对象的一个Observable
- `from()` — 将一个Iterable, 一个Future, 或者一个数组转换成一个Observable
- `repeat()` — 创建一个重复发射指定数据或数据序列的Observable
- `repeatWhen()` — 创建一个重复发射指定数据或数据序列的Observable，它依赖于另一个Observable发射的数据
- `create()` — 使用一个函数从头创建一个Observable
- `defer()` — 只有当订阅者订阅才创建Observable；为每个订阅创建一个新的Observable
- `range()` — 创建一个发射指定范围的整数序列的Observable
- `interval()` — 创建一个按照给定的时间间隔发射整数序列的Observable
- `timer()` — 创建一个在给定的延时之后发射单个数据的Observable
- `empty()` — 创建一个什么都不做直接通知完成的Observable
- `error()` — 创建一个什么都不做直接通知错误的Observable
- `never()` — 创建一个不发射任何数据的Observable

创建操作

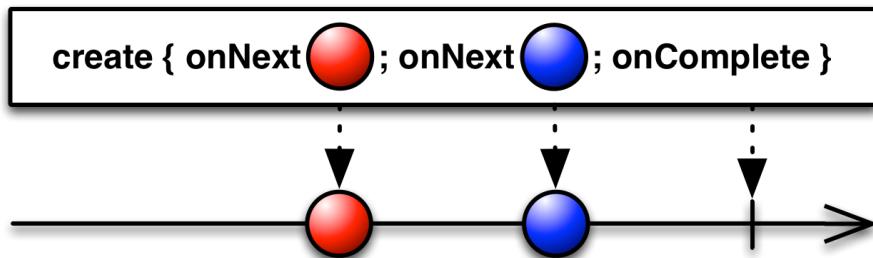
Create

使用一个函数从头开始创建一个Observable



你可以使用 `create` 操作符从头开始创建一个 Observable，给这个操作符传递一个接受观察者作为参数的函数，编写这个函数让它的行为表现为一个 Observable--恰当的调用观察者的 `onNext`, `onError` 和 `onCompleted` 方法。

一个形式正确的有限 Observable 必须尝试调用观察者的 `onCompleted` 正好一次或者它的 `onError` 正好一次，而且此后不能再调用观察者的任何其它方法。



RxJava 将这个操作符实现为 `create` 方法。

建议你在传递给 `create` 方法的函数中检查观察者的 `isUnsubscribed` 状态，以便在没有观察者的时候，让你的 Observable 停止发射数据或者做昂贵的运算。

示例代码：

```
observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> observer) {
        try {
            if (!observer.isUnsubscribed()) {
                for (int i = 1; i < 5; i++) {
                    observer.onNext(i);
                }
                observer.onCompleted();
            }
        } catch (Exception e) {
            observer.onError(e);
        }
    }
}).subscribe(new Subscriber<Integer>() {
    @Override
    public void onNext(Integer item) {
        System.out.println("Next: " + item);
    }
});
```

```

    }

    @Override
    public void onError(Throwable error) {
        System.err.println("Error: " +
error.getMessage());
    }

    @Override
    public void onCompleted() {
        System.out.println("Sequence complete.");
    }
);

```

输出:

```

Next: 1
Next: 2
Next: 3
Next: 4
Sequence complete.

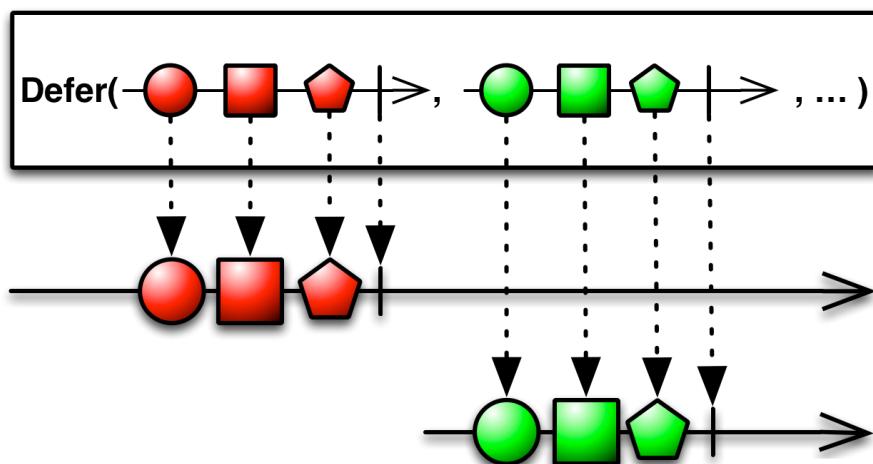
```

`create`方法默认不在任何特定的调度器上执行。

- Javadoc: [create\(OnSubscribe\)](#)

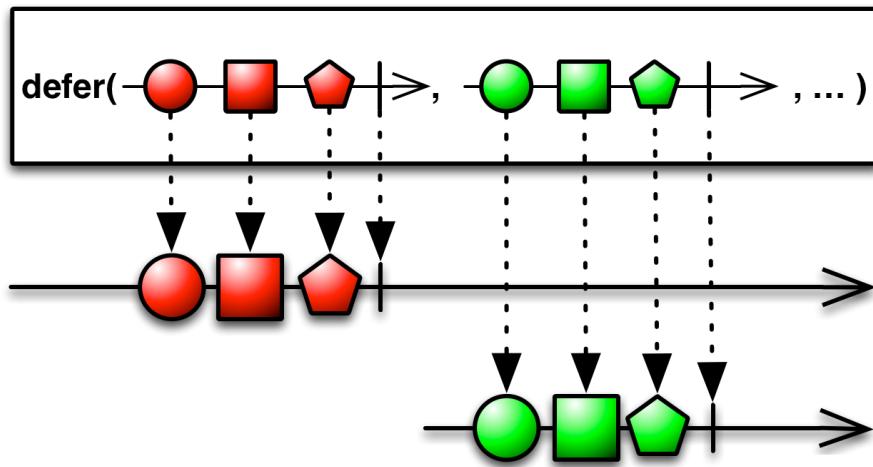
Defer

直到有观察者订阅时才创建Observable，并且为每个观察者创建一个新的Observable



`Defer`操作符会一直等待直到有观察者订阅它，然后它使用Observable工厂方法生成一个Observable。它对每个观察者都这样做，因此尽管每个订阅者都以为自己订阅的是同一个Observable，事实上每个订阅者获取的是它们自己的单独的数据序列。

在某些情况下，等待直到最后一分钟（就是知道订阅发生时）才生成Observable可以确保Observable包含最新的数据。

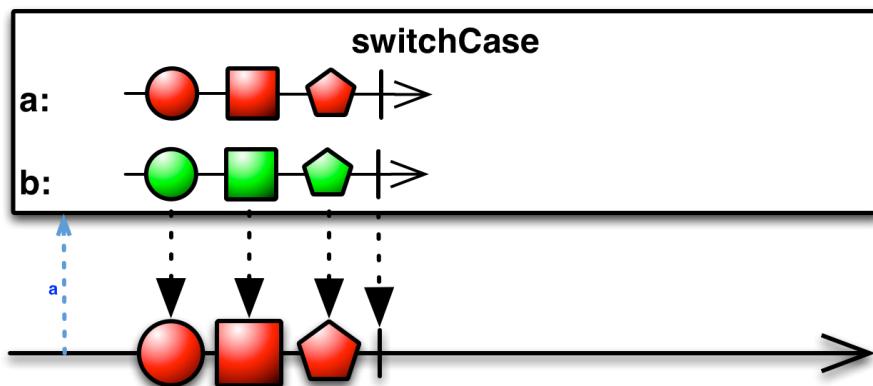


RxJava将这个操作符实现为 `defer` 方法。这个操作符接受一个你选择的 Observable工厂函数作为单个参数。这个函数没有参数，返回一个Observable。

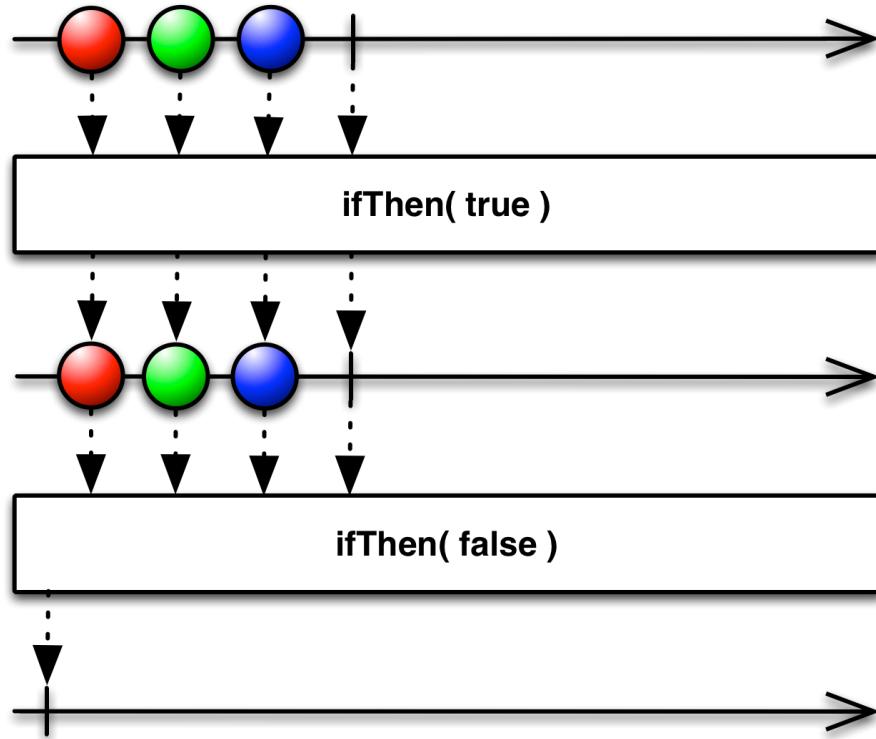
`defer`方法默认不在任何特定的调度器上执行。

- Javadoc: [defer\(Func0\)](#)

switchCase



可选包 [rxjava-computation-expressions](#) 中有一个类似的操作符。
`switchCase` 操作符有条件的创建并返回一个可能的Observables集合中的一个。



可选包 `rxjava-computation-expressions` 中还有一个更简单的操作符叫 `ifThen`。这个操作符检查某个条件，然后根据结果，返回原始Observable的镜像，或者返回一个空Observable。

Empty/Never/Throw

Empty

创建一个不发射任何数据但是正常终止的Observable

Never

创建一个不发射数据也不终止的Observable

Throw

创建一个不发射数据以一个错误终止的Observable

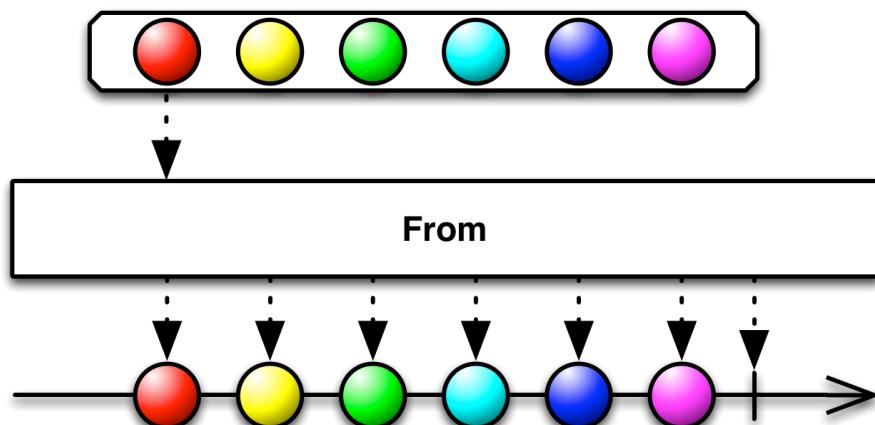
这三个操作符生成的Observable行为非常特殊和受限。测试的时候很有用，有时候也用于结合其它的Observables，或者作为其它需要Observable的操作符的参数。

RxJava将这些操作符实现为 `empty`, `never` 和 `error`。`error`操作符需要一个 `Throwable`参数，你的Observable会以此终止。这些操作符默认不在任何特定的调度器上执行，但是`empty`和`error`有一个可选参数是Scheduler，如果你传递了Scheduler参数，它们会在这个调度器上发送通知。

- Javadoc: `empty()`
- Javadoc: `never()`
- Javadoc: `error(java.lang.Throwable)`

From

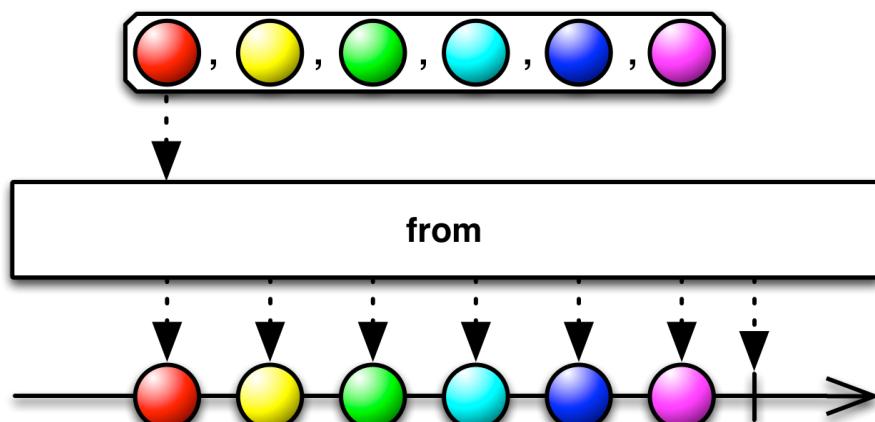
将其它种类的对象和数据类型转换为Observable



当你使用Observable时，如果你要处理的数据都可以转换成展现为Observables，而不是需要混合使用Observables和其它类型的数据，会非常方便。这让你在数据流的整个生命周期中，可以使用一组统一的操作符来管理它们。

例如，Iterable可以看成是同步的Observable；Future，可以看成是总是只发射单个数据的Observable。通过显式地将那些数据转换为Observables，你可以像使用Observable一样与它们交互。

因此，大部分ReactiveX实现都提供了将语言特定的对象和数据结构转换为Observables的方法。



在RxJava中，`from`操作符可以转换Future、Iterable和数组。对于Iterable和数组，产生的Observable会发射Iterable或数组的每一项数据。

示例代码

```
Integer[] items = { 0, 1, 2, 3, 4, 5 };
Observable myObservable = Observable.from(items);

myObservable.subscribe(
    new Action1<Integer>() {
        @Override
        public void call(Integer item) {
            System.out.println(item);
        }
    }
);
```

```
        }
    },
    new Action1<Throwable>() {
        @Override
        public void call(Throwable error) {
            System.out.println("Error encountered: " +
error.getMessage());
        }
    },
    new Action0() {
        @Override
        public void call() {
            System.out.println("Sequence complete");
        }
    }
);
```

输出

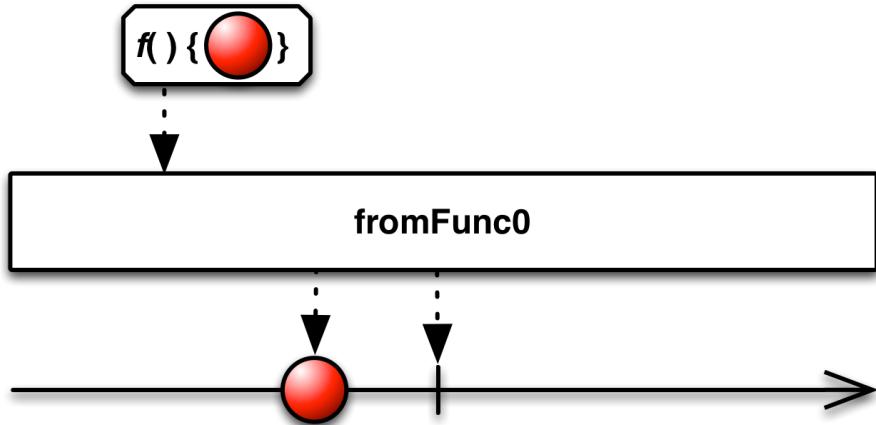
```
0
1
2
3
4
5
Sequence complete
```

对于Future，它会发射Future.get()方法返回的单个数据。`from`方法有一个可接受两个可选参数的版本，分别指定超时时长和时间单位。如果过了指定的时长Future还没有返回一个值，这个Observable会发射错误通知并终止。

`from`默认不在任何特定的调度器上执行。然而你可以将Scheduler作为可选的第二个参数传递给Observable，它会在那个调度器上管理这个Future。

- Javadoc: [from\(array\)](#)
- Javadoc: [from\(Iterable\)](#)
- Javadoc: [from\(Future\)](#)
- Javadoc: [from\(Future,Scheduler\)](#)
- Javadoc: [from\(Future,timeout, timeUnit\)](#)

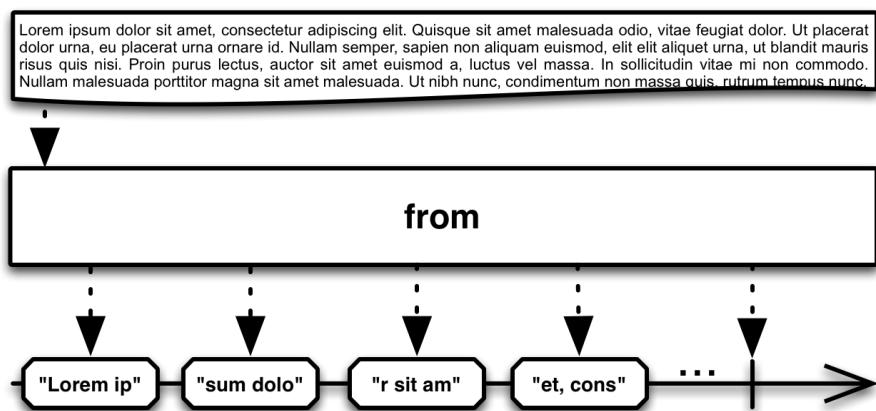
RxJavaAsyncUtil



此外，在可选包 `RxJavaAsyncUtil` 中，你还可以用下面这些操作符将actions, callables, functions和runnables转换为发射这些动作的执行结果的 Observable:

- `fromAction`
- `fromCallable`
- `fromFunc0`
- `fromRunnable`

在这个页面 [Start](#) 查看关于这些操作符的更多信息。



注意：还有一个可选的 `StringObservable` 类中也有一个 `from` 方法，它将一个字符流或者一个 `Reader` 转换为一个发射字节数组或字符串的 Observable。

runAsync2

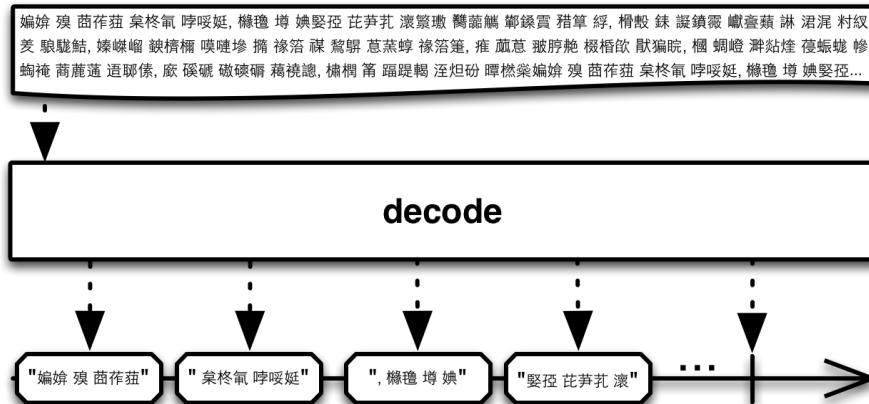
注意：这里与后面 `start` 操作符里的 `runAsync` 说明重复了

在单独的 `RxJavaAsyncUtil` 包中（默认不包含在 RxJava 中），还有一个 `runAsync` 函数。传递一个 `Action` 和一个 `Scheduler` 给 `runAsync`，它会返回一个 `StoppableObservable`，这个 Observable 使用 `Action` 产生发射的数据项。

传递一个 `Action` 和一个 `Scheduler` 给 `runAsync`，它返回一个使用这个 `Action` 产生数据的 `StoppableObservable`。这个 `Action` 接受一个 `Observable` 和一个 `Subscription` 作为参数，它使用 `Subscription` 检查 `unsubscribed` 条件，一旦发现条件为真就立即停止发射数据。任何时候你都可以使用 `unsubscribe` 方法手动停止一个 `StoppableObservable`（这会同时取消订阅与这个 `StoppableObservable` 关联的 `Subscription`）。

由于`runAsync`会立即调用`Action`并开始发射数据，在你创建`StoppableObservable`之后到你的观察者准备好接受数据之前这段时间里，可能会有一部分数据会丢失。如果这不符合你的要求，可以使用`runAsync`的一个变体，它也接受一个`Subject`参数，传递一个`ReplaySubject`给它，你可以获取其它丢失的数据了。

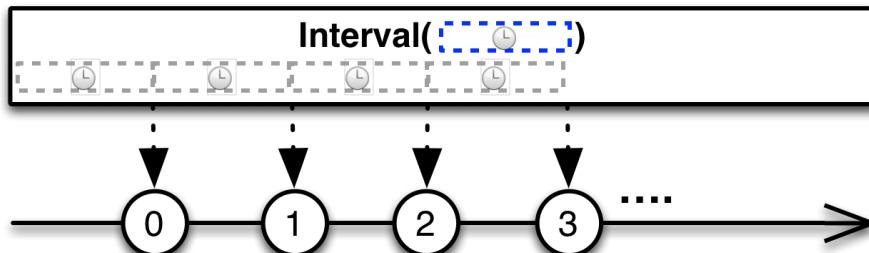
decode



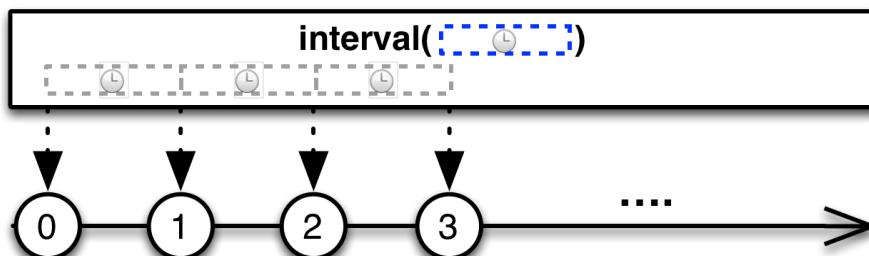
`StringObservable`类不是默认RxJava的一部分，包含一个`decode`操作符，这个操作符将一个多字节字符流转换为一个发射字节数组的Observable，这些字节数组按照字符的边界划分。

Interval

创建一个按固定时间间隔发射整数序列的Observable

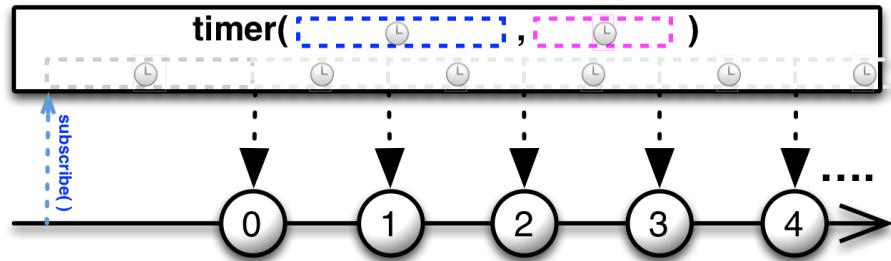


`Interval`操作符返回一个Observable，它按固定的时间间隔发射一个无限递增的整数序列。



RxJava将这个操作符实现为`interval`方法。它接受一个表示时间间隔的参数和一个表示时间单位的参数。

- Javadoc: `interval(long, TimeUnit)`
- Javadoc: `interval(long, TimeUnit, Scheduler)`



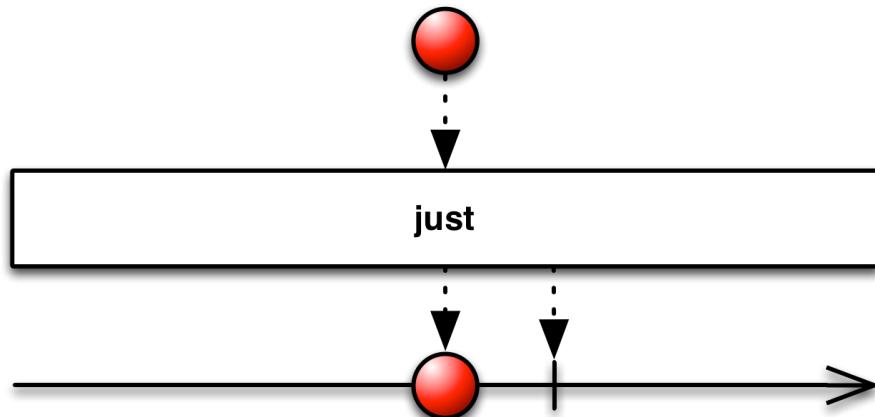
还有一个版本的 `interval` 返回一个 Observable，它在指定延迟之后先发射一个零值，然后再按照指定的时间间隔发射递增的数字。这个版本的 `interval` 在 RxJava 1.0.0 中叫做 `timer`，但是那个方法已经不建议使用了，因为一个名叫 `interval` 的操作符有同样的功能。

Javadoc: `interval(long,long,TimeUnit)` Javadoc:
`interval(long,long,TimeUnit,Scheduler)`

`interval` 默认在 `computation` 调度器上执行。你也可以传递一个可选的 `Scheduler` 参数来指定调度器。

Just

创建一个发射指定值的 Observable



Just 将单个数据转换为发射那个数据的 Observable。

Just 类似于 From，但是 From 会将数组或 Iterable 的数据取出然后逐个发射，而 Just 只是简单的原样发射，将数组或 Iterable 当做单个数据。

注意：如果你传递 `null` 给 Just，它会返回一个发射 `null` 值的 Observable。不要误认为它会返回一个空 Observable（完全不发射任何数据的 Observable），如果需要空 Observable 你应该使用 `Empty` 操作符。

RxJava 将这个操作符实现为 `just` 函数，它接受一至九个参数，返回一个按参数列表顺序发射这些数据的 Observable。

示例代码：

```
observable.just(1, 2, 3)
    .subscribe(new Subscriber<Integer>() {
        @Override
```

```

public void onNext(Integer item) {
    System.out.println("Next: " + item);
}

@Override
public void onError(Throwable error) {
    System.err.println("Error: " +
error.getMessage());
}

@Override
public void onCompleted() {
    System.out.println("Sequence complete.");
}
);

```

输出

```

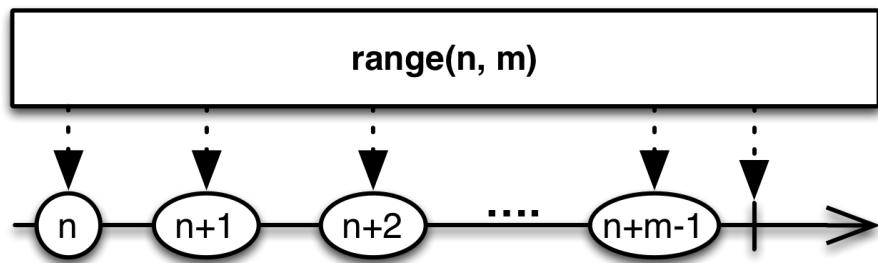
Next: 1
Next: 2
Next: 3
Sequence complete.

```

- Javadoc: [just\(item\)](#) (还有其它接受二到九个参数的版本)

Range

创建一个发射特定整数序列的Observable



Range操作符发射一个范围内的有序整数序列，你可以指定范围的起始和长度。

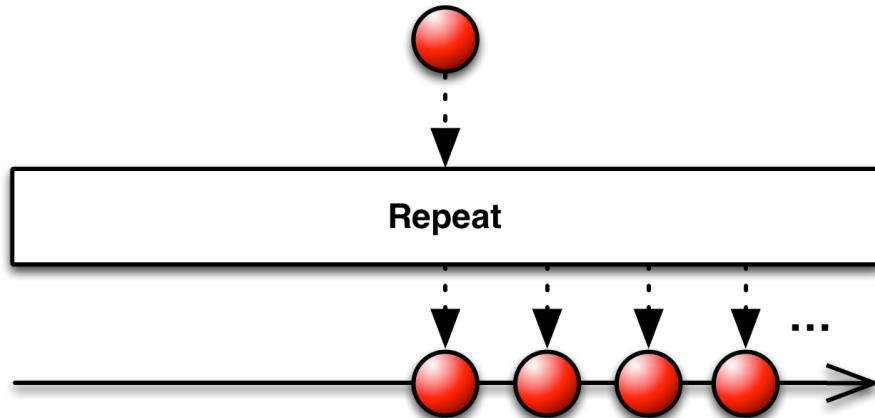
RxJava将这个操作符实现为 `range` 函数，它接受两个参数，一个是范围的起始值，一个是范围的数据的数目。如果你将第二个参数设为0，将导致Observable不发射任何数据（如果设置为负数，会抛异常）。

`range` 默认不在任何特定的调度器上执行。有一个变体可以通过可选参数指定 Scheduler。

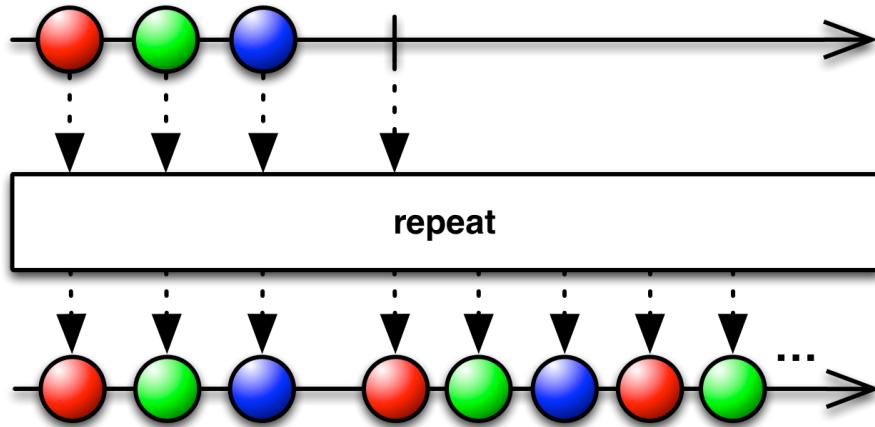
- Javadoc: [range\(int,int\)](#)
- Javadoc: [range\(int,int,Scheduler\)](#)

Repeat

创建一个发射特定数据重复多次的Observable



Repeat重复地发射数据。某些实现允许你重复的发射某个数据序列，还有一些允许你限制重复的次数。

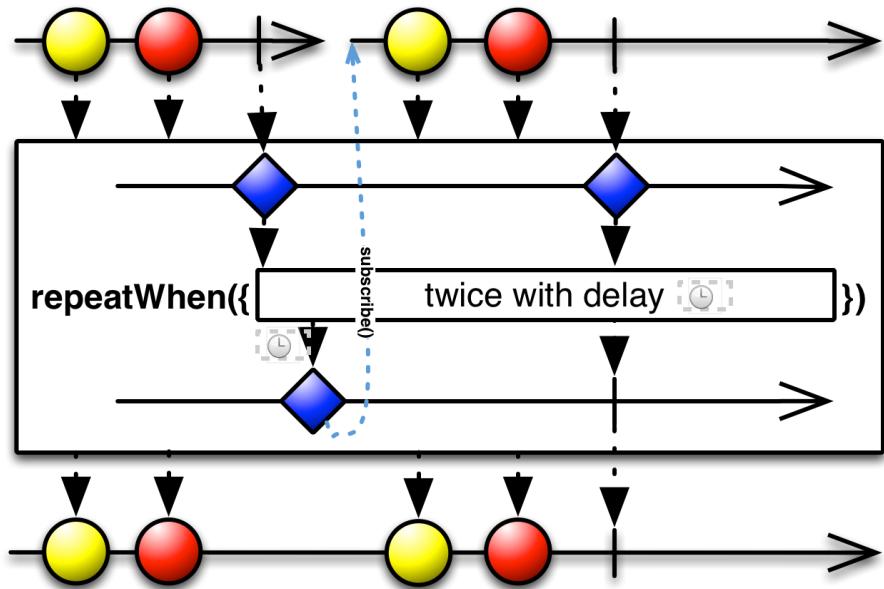


RxJava将这个操作符实现为 `repeat`方法。它不是创建一个Observable，而是重
复发射原始Observable的数据序列，这个序列或者是无限的，或者通过
`repeat(n)`指定重复次数。

`repeat`操作符默认在 `trampoline` 调度器上执行。有一个变体可以通过可选参数
指定Scheduler。

Javadoc: [repeat\(\)](#) Javadoc: [repeat\(long\)](#) Javadoc: [repeat\(Scheduler\)](#) Javadoc:
[repeat\(long,Scheduler\)](#)

repeatWhen



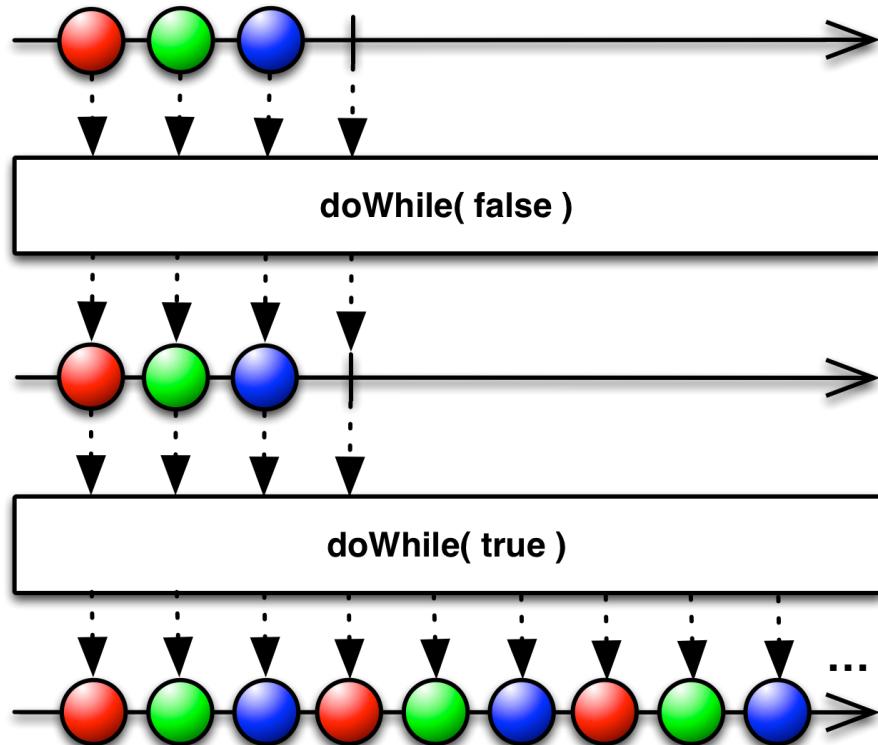
还有一个叫做 `repeatWhen` 的操作符，它不是缓存和重放原始Observable的数据序列，而是有条件的重新订阅和发射原来的Observable。

将原始Observable的终止通知（完成或错误）当做一个 `void` 数据传递给一个通知处理器，它以此来决定是否要重新订阅和发射原来的Observable。这个通知处理器就像一个Observable操作符，接受一个发射 `void` 通知的Observable为输入，返回一个发射 `void` 数据（意思是，重新订阅和发射原始Observable）或者直接终止（意思是，使用 `repeatwhen` 终止发射数据）的Observable。

`repeatwhen` 操作符默认在 `trampoline` 调度器上执行。有一个变体可以通过可选参数指定 `Scheduler`。

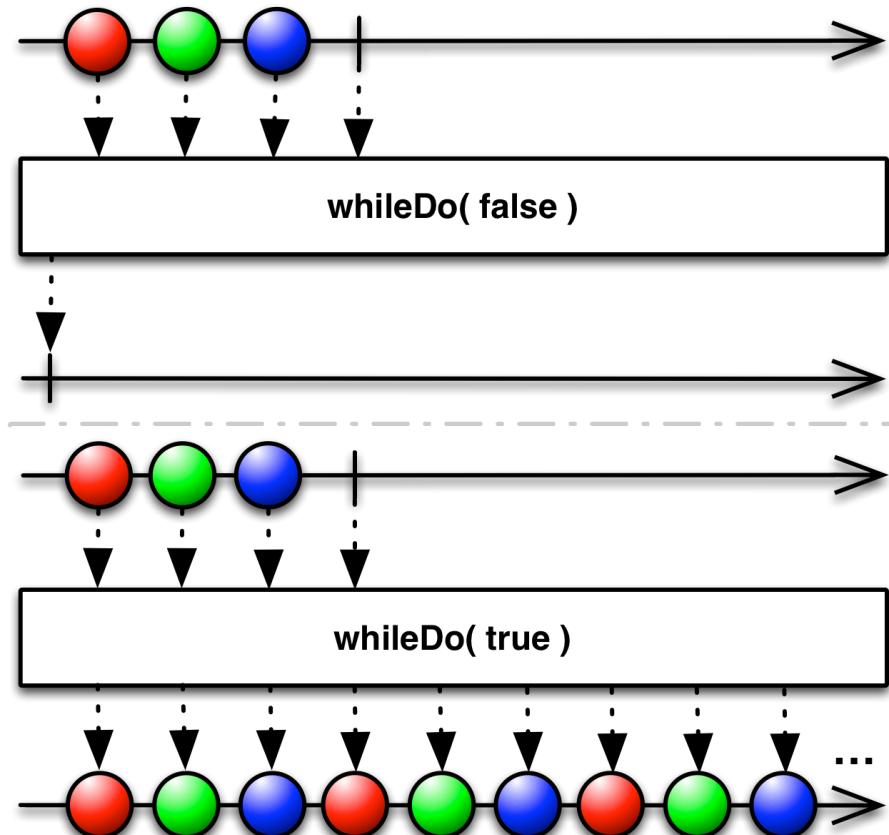
- Javadoc: [repeatWhen\(Func1\)](#)
- Javadoc: [repeatWhen\(Func1,Scheduler\)](#)

doWhile



`doWhile` 属于可选包 `rxjava-computation-expressions`, 不是RxJava标准操作符的一部分。`doWhile` 在原始序列的每次重复后检查某个条件, 如果满足条件才重复发射。

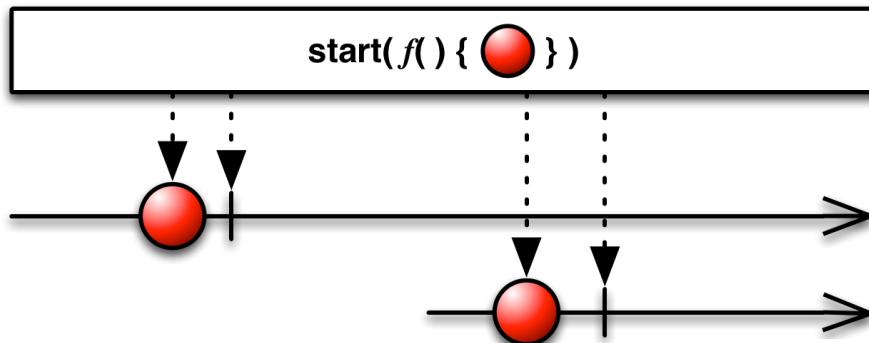
whileDo



`whileDo` 属于可选包 `rxjava-computation-expressions`, 不是RxJava标准操作符的一部分。`whileDo` 在原始序列的每次重复前检查某个条件, 如果满足条件才重复发射。

Start

返回一个Observable，它发射一个类似于函数声明的值



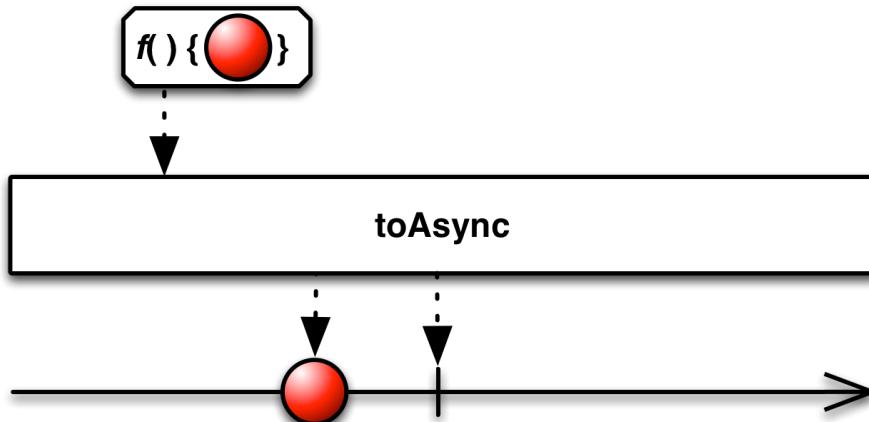
编程语言有很多种方法可以从运算结果中获取值，它们的名字一般叫 `functions`, `futures`, `actions`, `callables`, `runnables` 等等。在 `start` 目录下的这组操作符可以让它们表现得像 Observable，因此它们可以在 Observables 调用链中与其它 Observable 搭配使用。

`start` 操作符的多种 RxJava 实现都属于可选的 `rxjava-async` 模块。

`rxjava-async` 模块包含 `start` 操作符，它接受一个函数作为参数，调用这个函数获取一个值，然后返回一个会发射这个值给后续观察者的 Observable。

注意：这个函数只会被执行一次，即使多个观察者订阅这个返回的 Observable。

toAsync

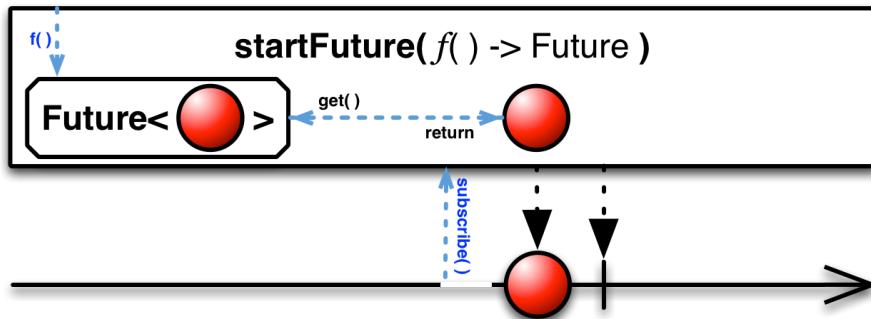


`rxjava-async` 模块还包含这几个操作符：`toAsync`, `asyncAction`, 和 `asyncFunc`。它们接受一个函数或一个 Action 作为参数。

对于函数(functions)，这个操作符调用这个函数获取一个值，然后返回一个会发射这个值给后续观察者的 Observable（和 `start` 一样）。对于动作(Action)，过程类似，但是没有返回值，在这种情况下，这个操作符在终止前会发射一个 `null` 值。

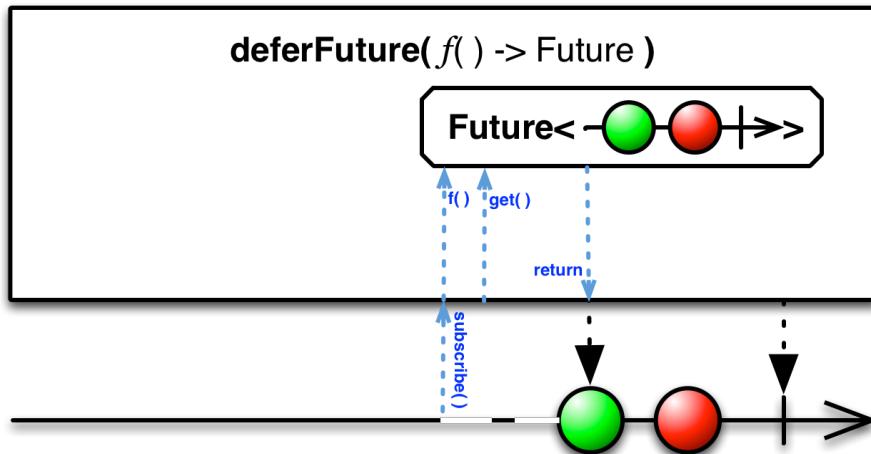
注意：这个函数或动作只会被执行一次，即使多个观察者订阅这个返回的 Observable。

startFuture



`rxjava-async` 模块还包含一个 `startFuture` 操作符，传递给它一个返回 `Future` 的函数，`startFuture` 会立即调用这个函数获取 `Future` 对象，然后调用 `Future` 的 `get()` 方法尝试获取它的值。它返回一个发射这个值给后续观察者的 Observable。

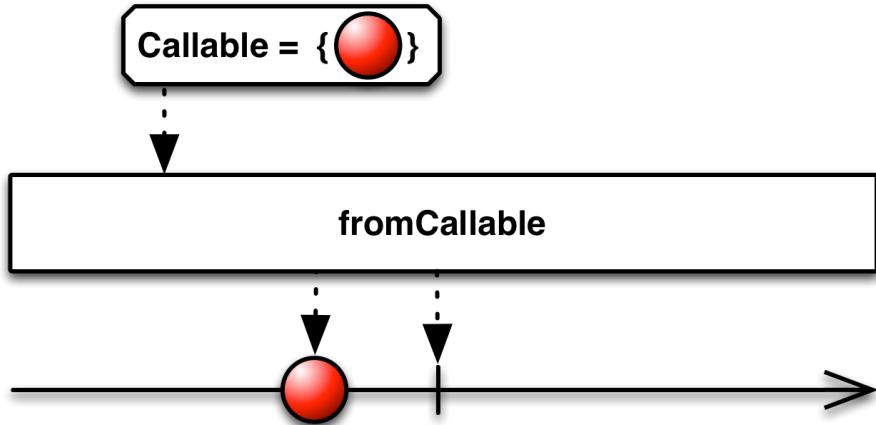
deferFuture



`rxjava-async` 模块还包含一个 `deferFuture` 操作符，传递给它一个返回 `Future` 的函数（这个 `Future` 返回一个 `Observable`），`deferFuture` 返回一个 `Observable`，但是不会调用你提供的函数，直到有观察者订阅它返回的 `Observable`。这时，它立即调用 `Future` 的 `get()` 方法，然后镜像发射 `get()` 方法返回的 `Observable` 发射的数据。

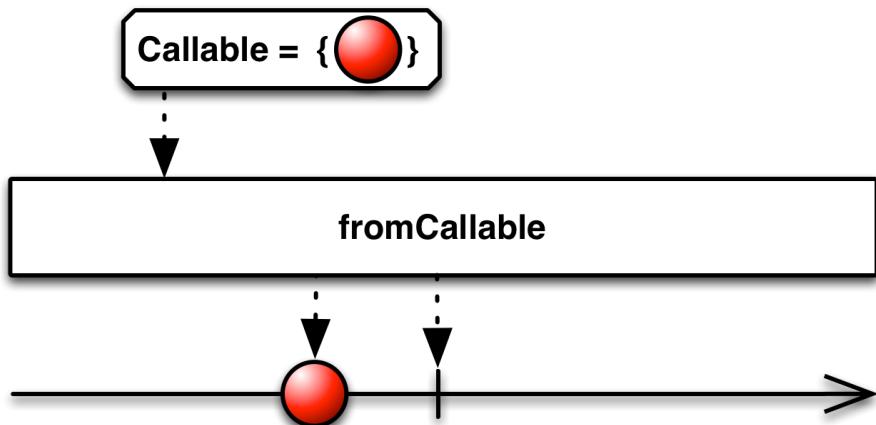
用这种方法，你可以在 Observables 调用链中包含一个返回 Observable 的 `Future` 对象。

fromAction



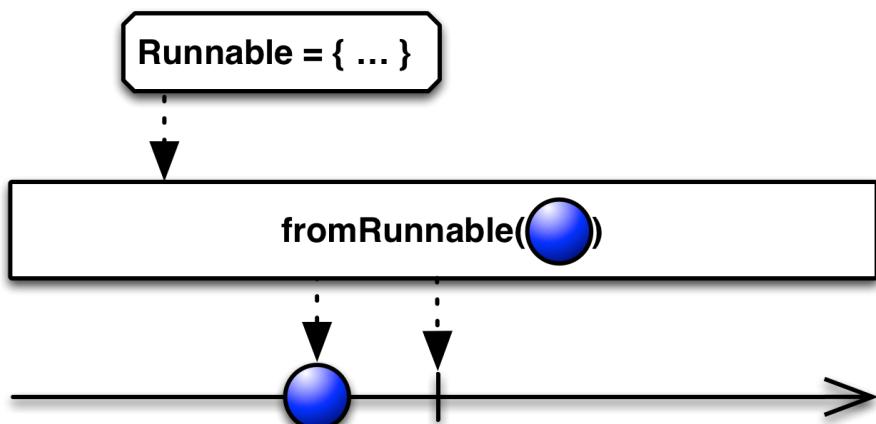
`rxjava-async` 模块还包含一个 `fromAction` 操作符，它接受一个 `Action` 作为参数，返回一个 `Observable`，一旦 `Action` 终止，它发射这个你传递给 `fromAction` 的数据。

fromCallable



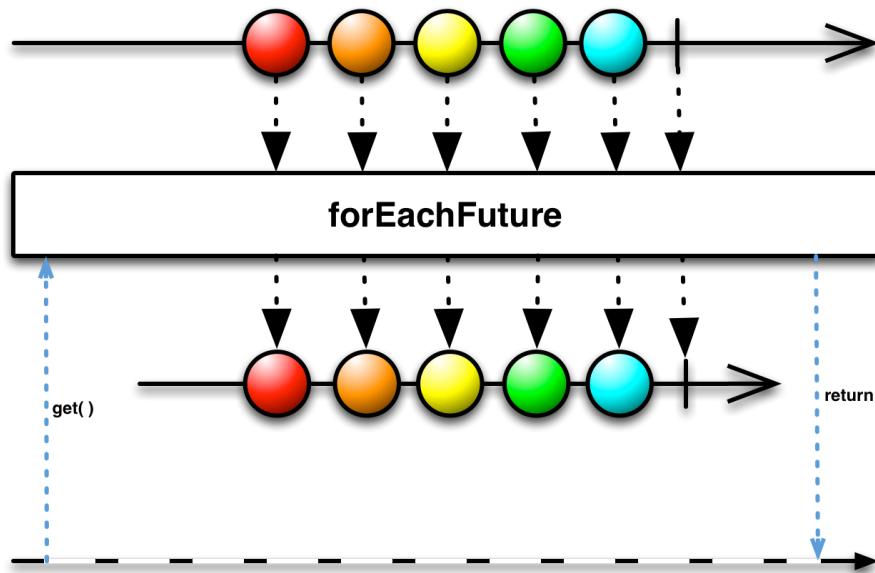
`rxjava-async` 模块还包含一个 `fromCallable` 操作符，它接受一个 `callable` 作为参数，返回一个发射这个 `callable` 的结果的 `Observable`。

fromRunnable



`rxjava-async` 模块还包含一个 `fromRunnable` 操作符，它接受一个 `Runnable` 作为参数，返回一个 `Observable`，一旦 `Runnable` 终止，它发射这个你传递给 `fromRunnable` 的数据。

forEachFuture



`rxjava-async` 模块还包含一个 `forEachFuture` 操作符。它其实不算 `start` 操作符的一个变体，而是有一些自己的特点。你传递一些典型的观察者方法（如 `onNext`, `onError` 和 `onCompleted`）给它， Observable 会以通常的方式调用它。但是 `forEachFuture` 自己返回一个 `Future` 并且在 `get()` 方法处阻塞，直到原始 Observable 执行完成，然后它返回，完成还是错误依赖于原始 Observable 是完成还是错误。

如果你想要一个函数阻塞直到 Observable 执行完成，可以使用这个操作符。

runAsync

`rxjava-async` 模块还包含一个 `runAsync` 操作符。它很特殊，返回一个叫做 `StoppableObservable` 的特殊 Observable。

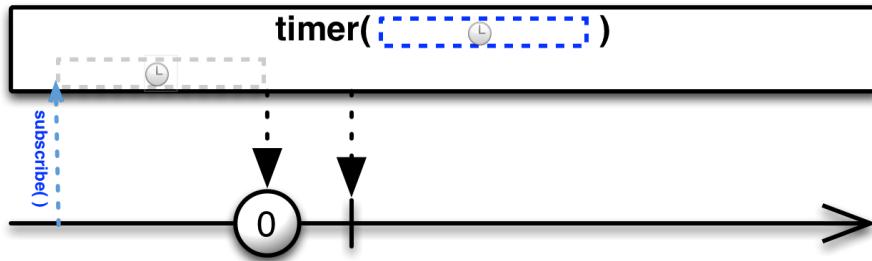
传递一个 `Action` 和一个 `Scheduler` 给 `runAsync`，它返回一个使用这个 `Action` 产生数据的 `StoppableObservable`。这个 `Action` 接受一个 `Observable` 和一个 `Subscription` 作为参数，它使用 `Subscription` 检查 `unsubscribed` 条件，一旦发现条件为真就立即停止发射数据。在任何时候你都可以使用 `unsubscribe` 方法手动停止一个 `StoppableObservable`（这会同时取消订阅与这个 `StoppableObservable` 关联的 `Subscription`）。

由于 `runAsync` 会立即调用 `Action` 并开始发射数据，在你创建 `StoppableObservable` 之后到你的观察者准备好接受数据之前这段时间里，可能会有一部分数据会丢失。如果这不符合你的要求，可以使用 `runAsync` 的一个变体，它也接受一个 `Subject` 参数，传递一个 `ReplaySubject` 给它，你可以获取其它丢失的数据了。

在 RxJava 中还有一个版本的 `From` 操作符可以将 Future 转换为 Observable，与 `start` 相似。

Timer

创建一个 Observable，它在一个给定的延迟后发射一个特殊的值。



`Timer`操作符创建一个在给定的时间段之后返回一个特殊值的Observable。

RxJava将这个操作符实现为`timer`函数。

`timer`返回一个Observable，它在延迟一段给定的时间后发射一个简单的数字0。

`timer`操作符默认在`computation`调度器上执行。有一个变体可以通过可选参数指定Scheduler。

- Javadoc: `timer(long,TimeUnit)`
- Javadoc: `timer(long,TimeUnit,Scheduler)`

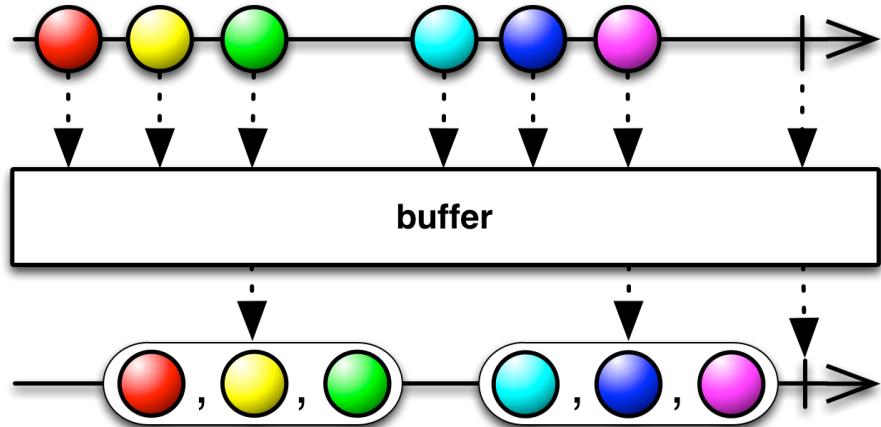
变换操作

这个页面展示了可用于对Observable发射的数据执行变换操作的各种操作符。

- **map()** – 对序列的每一项都应用一个函数来变换Observable发射的数据序列
- **flatMap(), concatMap(), and flatMapIterable()** – 将Observable发射的数据集合变换为Observables集合，然后将这些Observable发射的数据平坦化的放进一个单独的Observable
- **switchMap()** – 将Observable发射的数据集合变换为Observables集合，然后只发射这些Observables最近发射的数据
- **scan()** – 对Observable发射的每一项数据应用一个函数，然后按顺序依次发射每一个值
- **groupBy()** – 将Observable分拆为Observable集合，将原始Observable发射的数据按Key分组，每一个Observable发射一组不同的数据
- **buffer()** – 它定期从Observable收集数据到一个集合，然后把这些数据集合打包发射，而不是一次发射一个
- **window()** – 定期将来自Observable的数据分拆成一些Observable窗口，然后发射这些窗口，而不是每次发射一项
- **cast()** – 在发射之前强制将Observable发射的所有数据转换为指定类型

Buffer

定期收集Observable的数据放进一个数据包裹，然后发射这些数据包裹，而不是一次发射一个值。



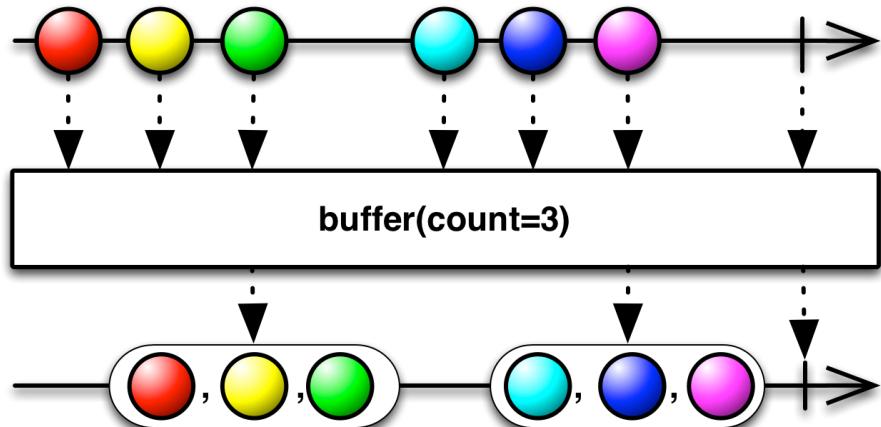
`Buffer`操作符将一个Observable变換为另一个，原来的Observable正常发射数据，变換产生的Observable发射这些数据的缓存集合。`Buffer`操作符在很多语言特定的实现中有很多种变体，它们在如何缓存这个问题上存在区别。

注意：如果原来的Observable发射了一个`onError`通知，`Buffer`会立即传递这个通知，而不是首先发射缓存的数据，即使在这之前缓存中包含了原始Observable发射的数据。

`window`操作符与`Buffer`类似，但是它在发射之前把收集到的数据放进单独的Observable，而不是放进一个数据结构。

在RxJava中有许多`Buffer`的变体：

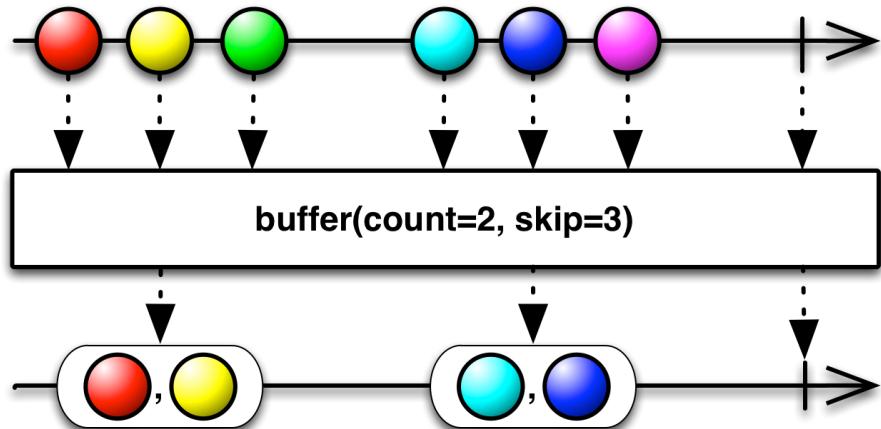
`buffer(count)`



`buffer(count)`以列表(List)的形式发射非重叠的缓存，每一个缓存至多包含来自原始Observable的count项数据（最后发射的列表数据可能少于count项）

- Javadoc: [buffer\(int\)](#)

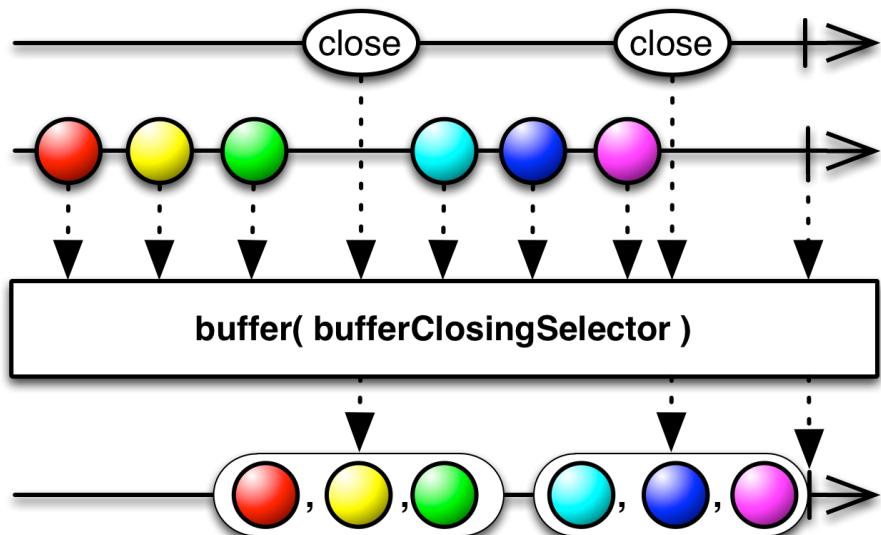
`buffer(count, skip)`



`buffer(count, skip)` 从原始Observable的第一项数据开始创建新的缓存，此后每当收到`skip`项数据，用`count`项数据填充缓存：开头的一项和后续的`count-1`项，它以列表(List)的形式发射缓存，取决于`count`和`skip`的值，这些缓存可能会有重叠部分（比如`skip < count`时），也可能会有间隙（比如`skip > count`时）。

- Javadoc: [buffer\(int,int\)](#)

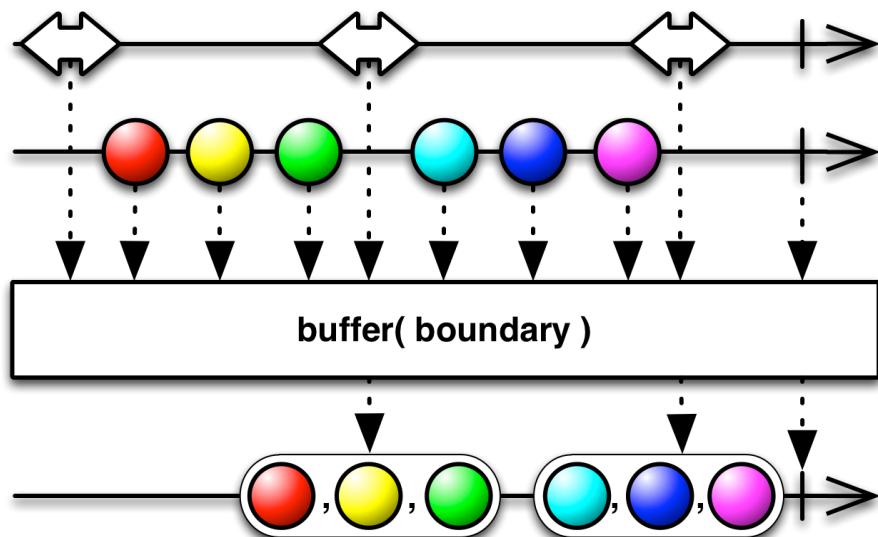
buffer(bufferClosingSelector)



当它订阅原来的Observable时，`buffer(bufferClosingSelector)`开始将数据收集到一个List，然后它调用`bufferClosingSelector`生成第二个Observable，当第二个Observable发射一个`Tclosing`时，`buffer`发射当前的List，然后重复这个过程：开始组装一个新的List，然后调用`bufferClosingSelector`创建一个新的Observable并监视它。它会一直这样做直到原来的Observable执行完成。

- Javadoc: [buffer\(Func0\)](#)

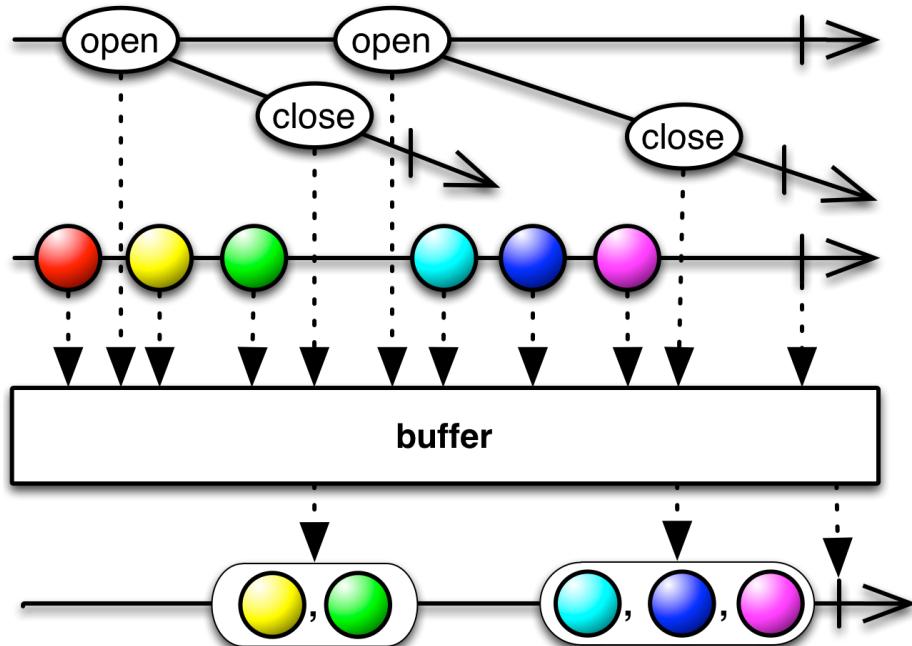
buffer(boundary)



`buffer(boundary)` 监视一个名叫 `boundary` 的 Observable，每当这个 Observable 发射了一个值，它就创建一个新的 List 开始收集来自原始 Observable 的数据并发射原来的 List。

- Javadoc: `buffer(Observable)`
- Javadoc: `buffer(Observable,int)`

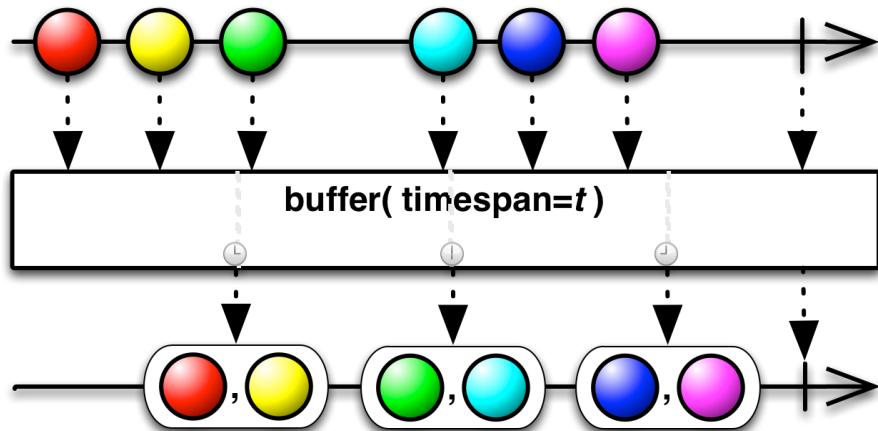
`buffer(bufferOpenings, bufferClosingSelector)`



`buffer(bufferOpenings, bufferClosingSelector)` 监视这个叫 `bufferOpenings` 的 Observable（它发射 `BufferOpening` 对象），每当 `bufferopenings` 发射了一个数据时，它就创建一个新的 List 开始收集原始 Observable 的数据，并将 `bufferopenings` 传递给 `closingselector` 函数。这个函数返回一个 Observable。`buffer` 监视这个 Observable，当它检测到一个来自这个 Observable 的数据时，就关闭 List 并且发射它自己的数据（之前的那个 List）。

- Javadoc: `buffer(Observable,Func1)`

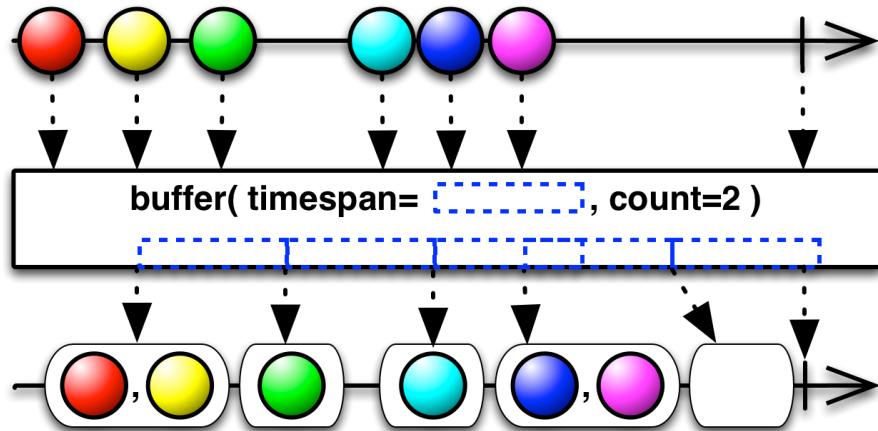
`buffer(timespan, unit[, scheduler])`



`buffer(timespan, unit)` 定期以 `List` 的形式发射新的数据，每个时间段，收集来自原始 `Observable` 的数据（从前面一个数据包裹之后，或者如果是第一个数据包裹，从有观察者订阅原来的 `Observale` 之后开始）。还有另一个版本的 `buffer` 接受一个 `Scheduler` 参数，默认情况下会使用 `computation` 调度器。

- Javadoc: [buffer\(long, TimeUnit\)](#)
- Javadoc: [buffer\(long, TimeUnit, Scheduler\)](#)

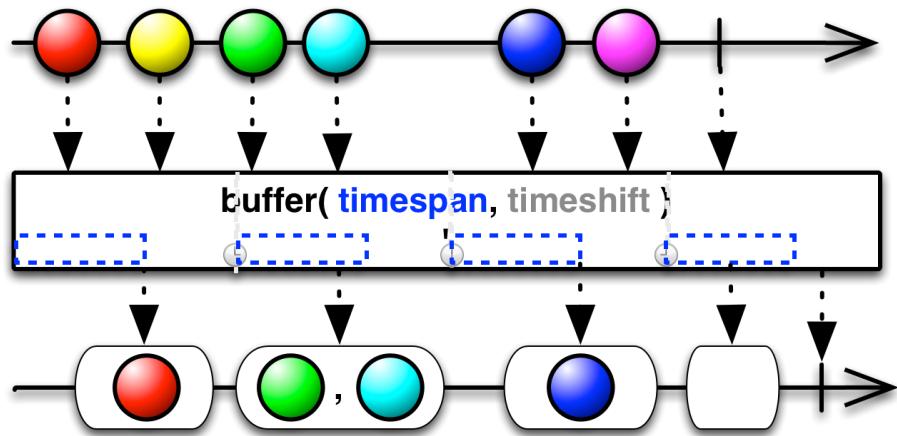
`buffer(timespan, unit, count[, scheduler])`



每当收到来自原始 `Observable` 的 `count` 项数据，或者每过了一段指定的时间后，`buffer(timespan, unit, count)` 就以 `List` 的形式发射这期间的数据，即使数据项少于 `count` 项。还有另一个版本的 `buffer` 接受一个 `Scheduler` 参数，默认情况下会使用 `computation` 调度器。

- Javadoc: [buffer\(long, TimeUnit, int\)](#)
- Javadoc: [buffer\(long, TimeUnit, int, Scheduler\)](#)

`buffer(timespan, timeshift, unit[, scheduler])`



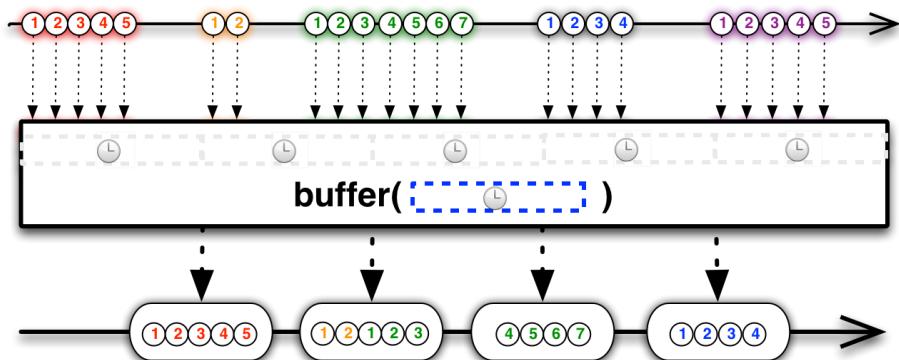
`buffer(timespan, timeshift)` 在每一个 `timeshift` 时期内都创建一个 `List`, 然后用原始 Observable 发射的每一项数据填充这个列表 (在把这个 `List` 当做自己的数据发射前, 从创建时开始, 直到过了 `timespan` 这么长的时间)。如果 `timespan` 长于 `timeshift`, 它发射的数据包将会重叠, 因此可能包含重复的数据项。

还有另一个版本的 `buffer` 接受一个 `Scheduler` 参数, 默认情况下会使用 `computation` 调度器。

- Javadoc: `buffer(long, long, TimeUnit)`
- Javadoc: `buffer(long, long, TimeUnit, Scheduler)`

buffer-backpressure

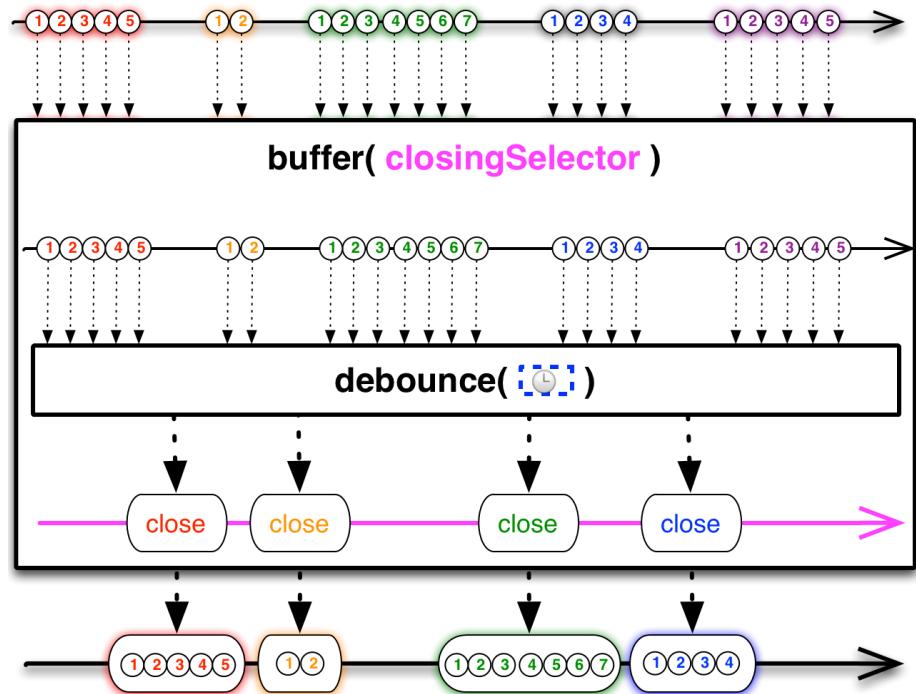
你可以使用 `Buffer` 操作符实现反压 `backpressure` (意思是, 处理这样一个 Observable: 它产生数据的速度可能比它的观察者消费数据的速度快)。



Buffer 操作符可以将大量的数据序列缩减为较少的数据缓存序列, 让它们更容易处理。例如, 你可以按固定的时间间隔, 定期关闭和发射来自一个爆发性 Observable 的数据缓存。这相当于一个缓冲区。

示例代码

```
Observable<List<Integer>> burstyBuffered =
    bursty.buffer(500, TimeUnit.MILLISECONDS);
```



或者，如果你想更进一步，可以在爆发期将数据收集到缓存，然后在爆发期终止时发射这些数据，使用 `Debounce` 操作符给 `buffer` 操作符发射一个缓存关闭指示器(`buffer closing indicator`)可以做到这一点。

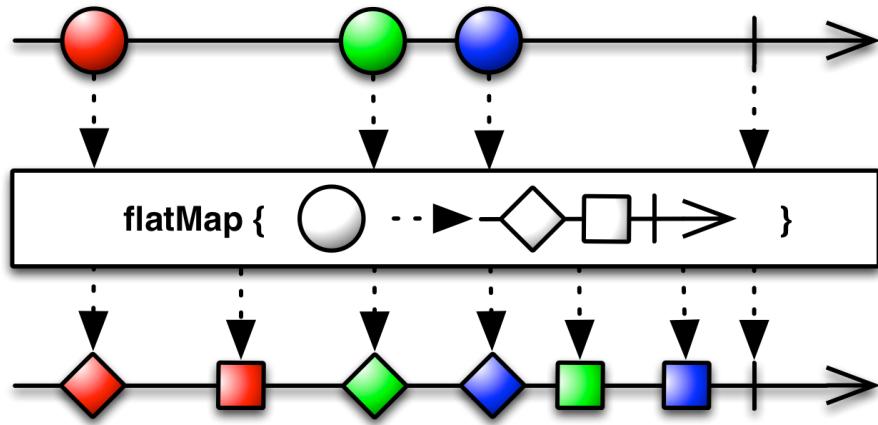
代码示例：

```
// we have to multicast the original bursty observable so
we can use it
// both as our source and as the source for our buffer
closing selector:
Observable<Integer> burstyMulticast =
bursty.publish().refCount();
// burstyDebounced will be our buffer closing selector:
Observable<Integer> burstyDebounced =
burstyMulticast.debounce(10, TimeUnit.MILLISECONDS);
// and this, finally, is the Observable of buffers we're
interested in:
Observable<List<Integer>> burstyBuffered =
burstyMulticast.buffer(burstyDebounced);
```

参见

FlatMap

`FlatMap` 将一个发射数据的 Observable 变换为多个 Observables，然后将它们发射的数据合并后放进一个单独的 Observable

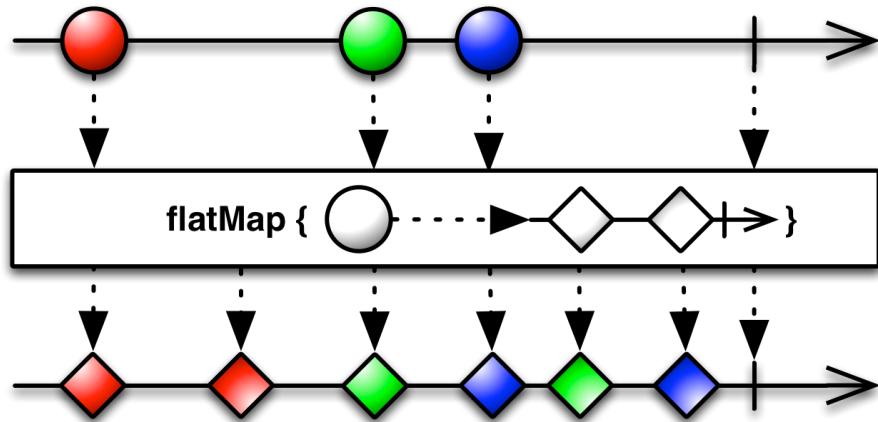


`flatMap`操作符使用一个指定的函数对原始Observable发射的每一项数据执行变换操作，这个函数返回一个本身也发射数据的Observable，然后`flatMap`合并这些Observables发射的数据，最后将合并后的结果当做它自己的数据序列发射。

这个方法是很有用的，例如，当你有一个这样的Observable：它发射一个数据序列，这些数据本身包含Observable成员或者可以变换为Observable，因此你可以创建一个新的Observable发射这些次级Observable发射的数据的完整集合。

注意：`flatMap`对这些Observables发射的数据做的是合并(`merge`)操作，因此它们可能是交错的。

在许多语言特定的实现中，还有一个操作符不会让变换后的Observables发射的数据交错，它按照严格的顺序发射这些数据，这个操作符通常被叫作`ConcatMap`或者类似的名字。

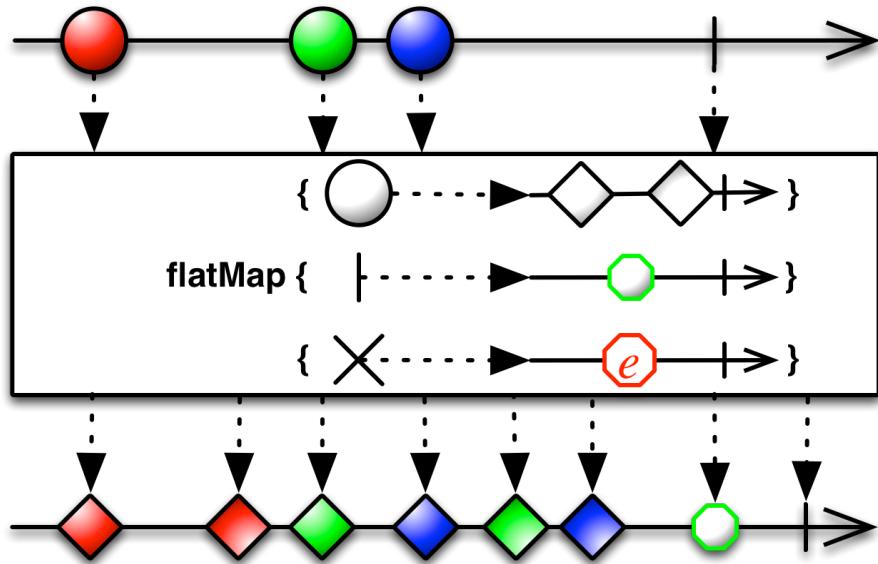


RxJava将这个操作符实现为`flatMap`函数。

注意：如果任何一个通过这个`flatMap`操作产生的单独的Observable调用`onError`异常终止了，这个Observable自身会立即调用`onError`并终止。

这个操作符有一个接受额外的`int`参数的一个变体。这个参数设置`flatMap`从原来的Observable映射Observables的最大同时订阅数。当达到这个限制时，它会等待其中一个终止然后再订阅另一个。

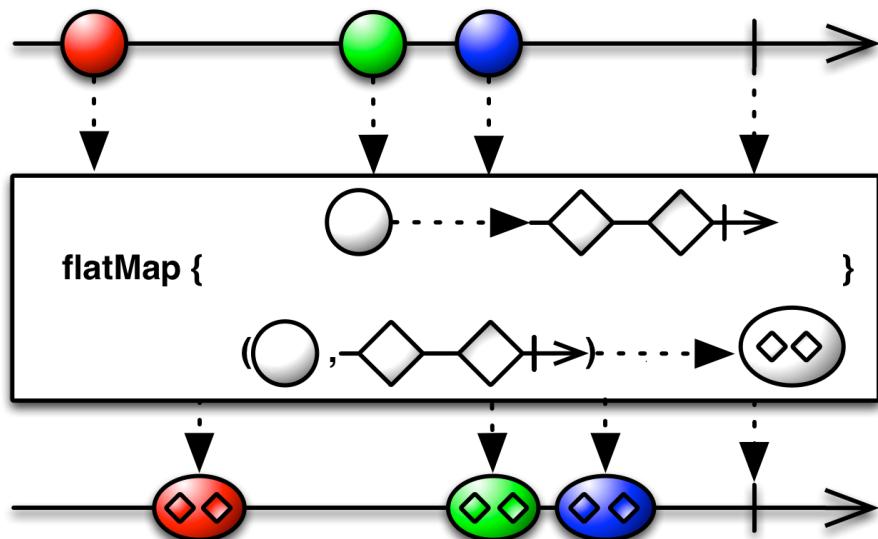
- Javadoc: [flatMap\(Func1\)](#)
- Javadoc: [flatMap\(Func1,int\)](#)



还有一个版本的 `flatMap` 为原始 Observable 的每一项数据和每一个通知创建一个新 的 Observable（并对数据平坦化）。

它也有一个接受额外 `int` 参数的变体。

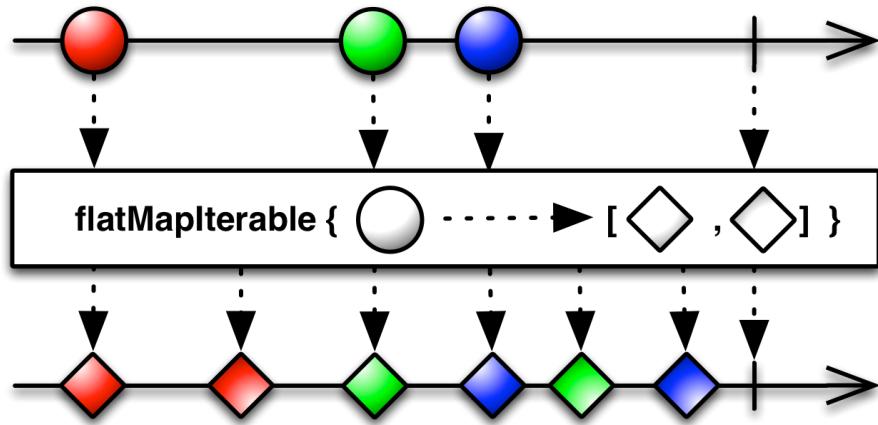
- Javadoc: `flatMap(Func1,Func1,Func0)`
- Javadoc: `flatMap(Func1,Func1,Func0,int)`



还有一个版本的 `flatMap` 会使用原始 Observable 的数据触发的 Observable 组合这些数据，然后发射这些数据组合。它也有一个接受额外 `int` 参数的版本。

- Javadoc: `flatMap(Func1,Func2)`
- Javadoc: `flatMap(Func1,Func2,int)`

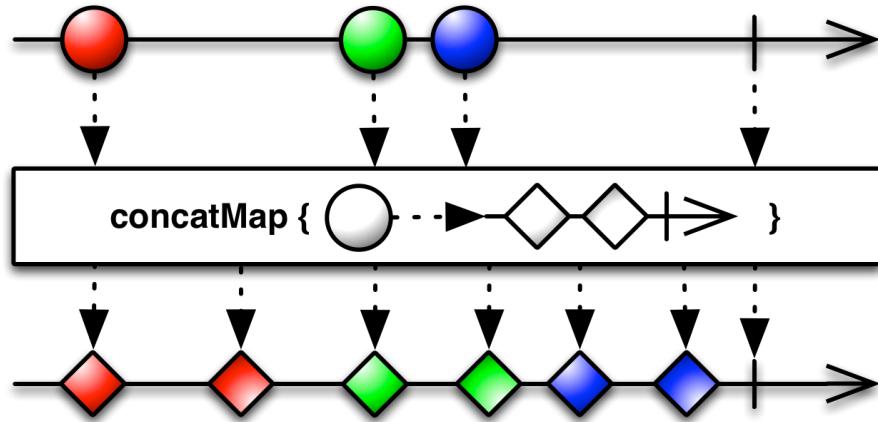
flatMapIterable



`flatMapIterable`这个变体成对的打包数据，然后生成Iterable而不是原始数据和生成的Observables，但是处理方式是相同的。

- Javadoc: [flatMapIterable\(Func1\)](#)
- Javadoc: [flatMapIterable\(Func1,Func2\)](#)

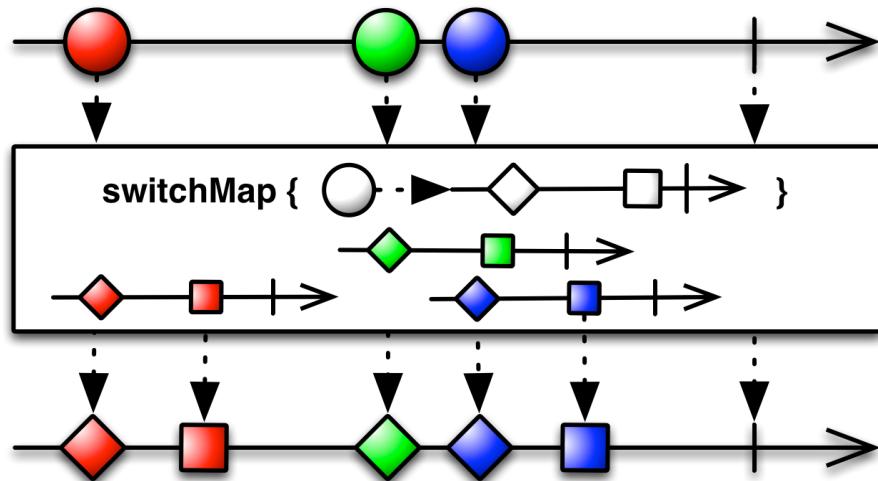
concatMap



还有一个`concatMap`操作符，它类似于最简单版本的`flatMap`，但是它按次序连接而不是合并那些生成的Observables，然后产生自己的数据序列。

- Javadoc: [concatMap\(Func1\)](#)

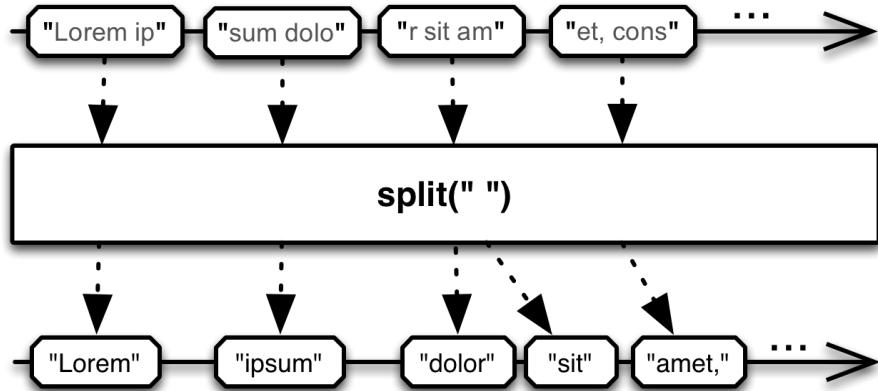
switchMap



RxJava还实现了 `switchMap` 操作符。它和 `flatMap` 很像，除了一点：当原始 Observable 发射一个新的数据（Observable）时，它将取消订阅并停止监视产生执之前那个数据的 Observable，只监视当前这一个。

- Javadoc: `switchMap(Func1)`

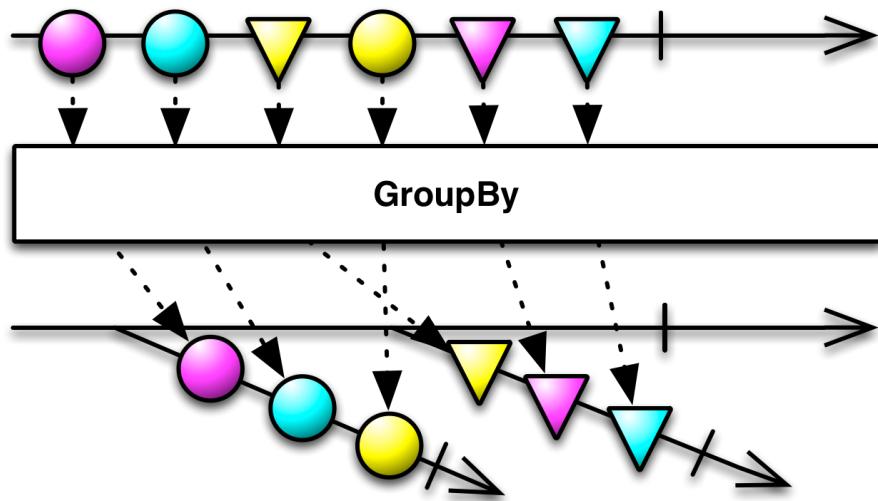
split



在特殊的 `StringObservable` 类（默认没有包含在 RxJava 中）中还有一个 `split` 操作符。它将一个发射字符串的 `Observable` 转换为另一个发射字符串的 `Observable`，只不过，后者将原始的数据序列当做一个数据流，使用一个正则表达式边界分割它们，然后合并发射分割的结果。

GroupBy

将一个Observable分拆为一些Observables集合，它们中的每一个发射原始 Observable的一个子序列



GroupBy 操作符将原始Observable分拆为一些Observables集合，它们中的每一个发射原始Observable数据序列的一个子序列。哪个数据项由哪一个Observable发射是由一个函数判定的，这个函数给每一项指定一个Key，Key相同的数据会被同一个Observable发射。

RxJava实现了`groupBy`操作符。它返回Observable的一个特殊子类`GroupedObservable`，实现了`GroupedObservable`接口的对象有一个额外的方法`getKey`，这个Key用于将数据分组到指定的Observable。

有一个版本的 `groupBy` 允许你传递一个变换函数，这样它可以在发射结果 `GroupedObservable` 之前改变数据项。

注意： `groupBy` 将原始 `Observable` 分解为一个发射多个 `GroupedObservable` 的 `Observable`，一旦有订阅，每个 `GroupedObservable` 就开始缓存数据。因此，如果你忽略这些 `GroupedObservable` 中的任何一个，这个缓存可能形成一个潜在的内存泄露。因此，如果你不想观察，也不要忽略 `GroupedObservable`。你应该使用像 `take(0)` 这样会丢弃自己的缓存的操作符。

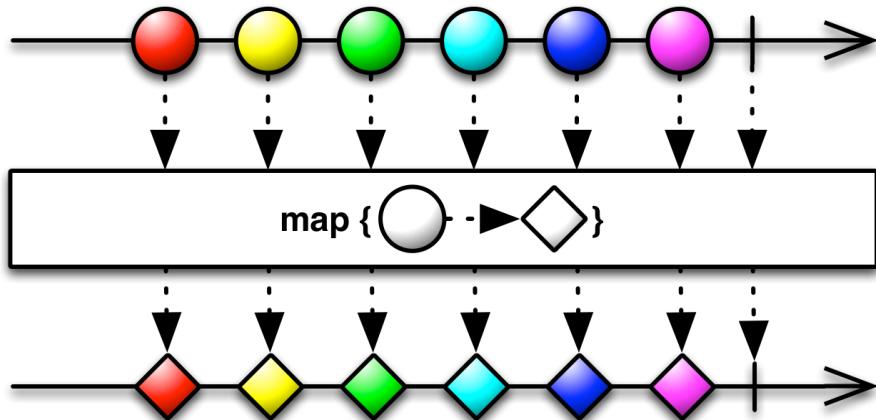
如果你取消订阅一个 `GroupedObservable`，那个 `Observable` 将会终止。如果之后原始的 `Observable` 又发射了一个与这个 `Observable` 的 Key 匹配的数据，`groupBy` 将会为这个 Key 创建一个新的 `GroupedObservable`。

`groupBy` 默认不在任何特定的调度器上执行。

- Javadoc: [groupBy\(Func1\)](#)
- Javadoc: [groupBy\(Func1,Func1\)](#)

Map

对 `Observable` 发射的每一项数据应用一个函数，执行变换操作

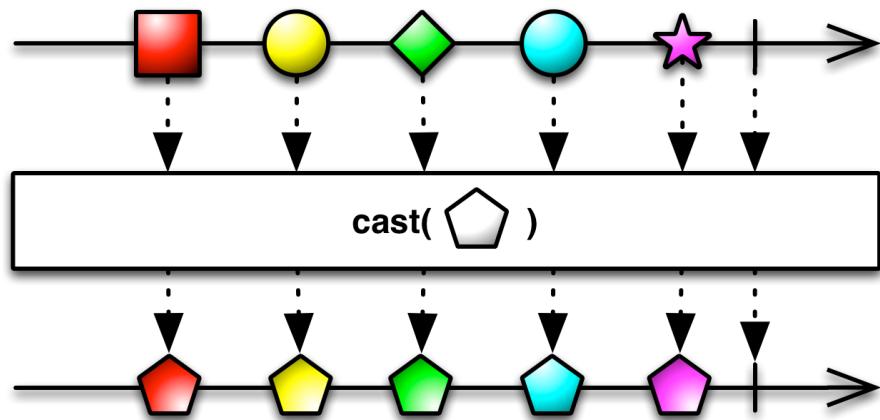


`Map` 操作符对原始 `Observable` 发射的每一项数据应用一个你选择的函数，然后返回一个发射这些结果的 `Observable`。

RxJava 将这个操作符实现为 `map` 函数。这个操作符默认不在任何特定的调度器上执行。

- Javadoc: [map\(Func1\)](#)

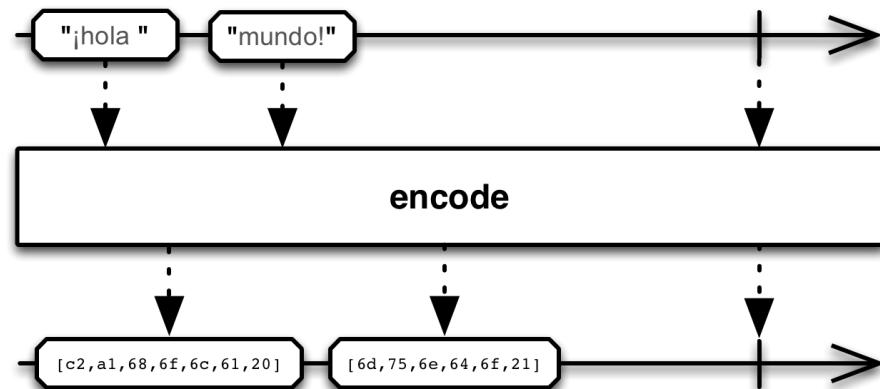
cast



`cast`操作符将原始Observable发射的每一项数据都强制转换为一个指定的类型，然后再发射数据，它是`map`的一个特殊版本。

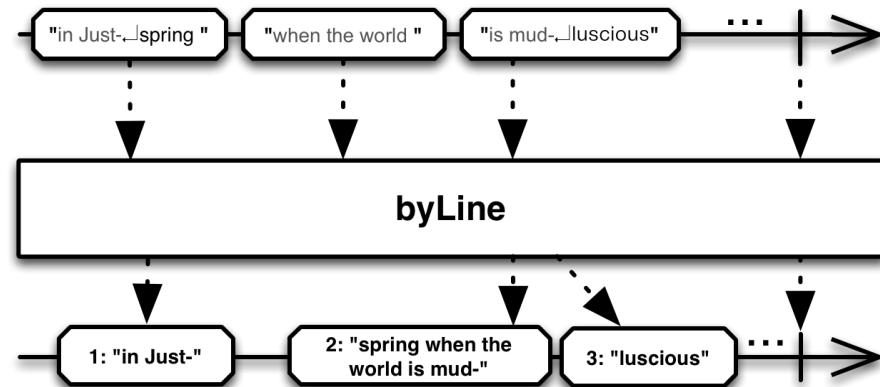
- Javadoc: [cast\(Class\)](#)

encode



`encode`在`StringObservable`类中，不是标准RxJava的一部分，它也是一个特殊的`map`操作符。`encode`将一个发射字符串的Observable变换为一个发射字节数组（这个字节数组按照原始字符串中的多字节字符边界划分）的Observable。

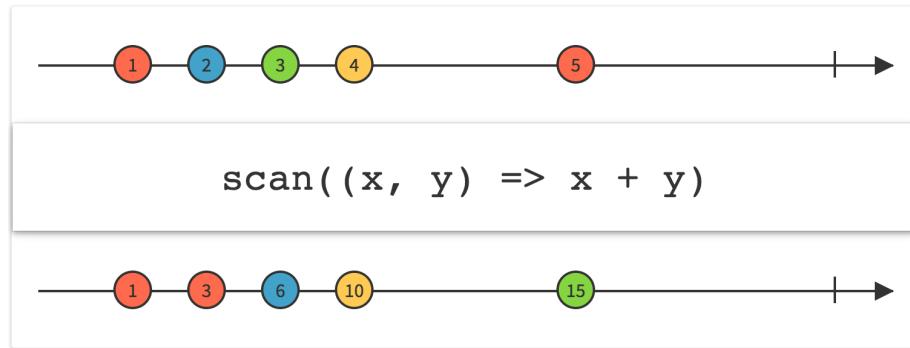
byLine



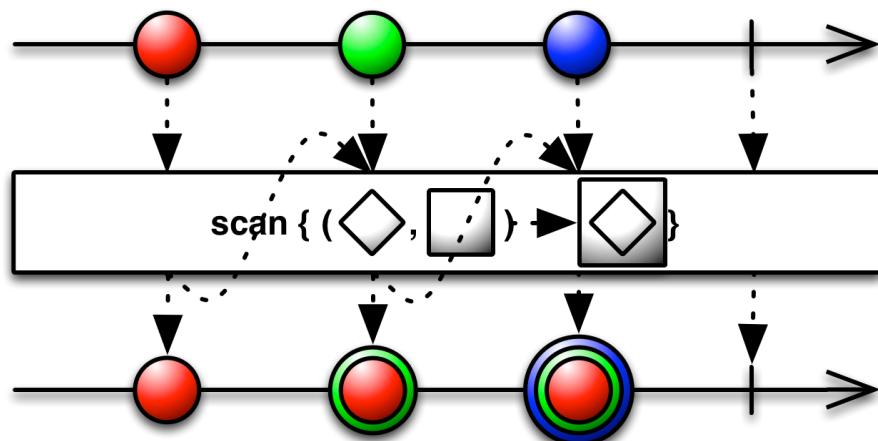
`byLine`同样在`StringObservable`类中，也不是标准RxJava的一部分，它也是一个特殊的`map`操作符。`byLine`将一个发射字符串的Observable变换为一个按行发射来自原始Observable的字符串的Observable。

Scan

连续地对数据序列的每一项应用一个函数，然后连续发射结果



Scan操作符对原始Observable发射的第一项数据应用一个函数，然后将那个函数的结果作为自己的第一项数据发射。它将函数的结果同第二项数据一起填充给这个函数来产生它自己的第二项数据。它持续进行这个过程来产生剩余的数据序列。这个操作符在某些情况下被叫做 **accumulator**。



RxJava实现了**scan**操作符。

示例代码：

```
Observable.just(1, 2, 3, 4, 5)  
    .scan(new Func2<Integer, Integer, Integer>() {  
        @Override  
        public Integer call(Integer sum, Integer item) {  
            return sum + item;  
        }  
    }).subscribe(new Subscriber<Integer>() {  
        @Override  
        public void onNext(Integer item) {  
            System.out.println("Next: " + item);  
        }  
  
        @Override  
        public void onError(Throwable error) {  
            System.err.println("Error: " +  
                error.getMessage());  
        }  
    })
```

```

@Override
public void onCompleted() {
    System.out.println("Sequence complete.");
}
);

```

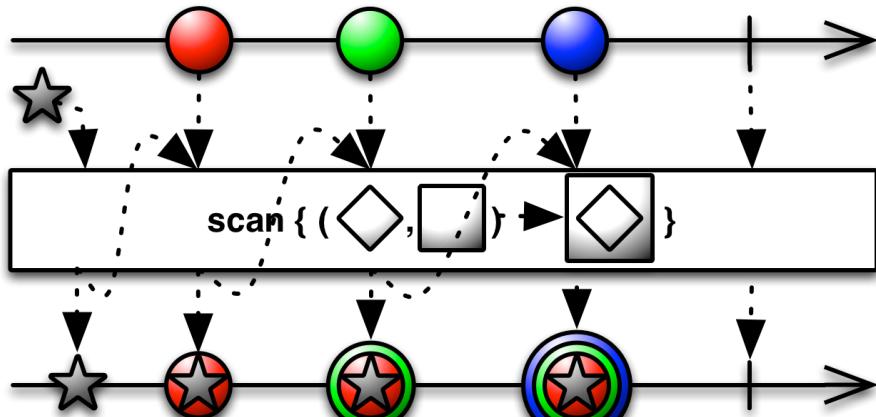
输出

```

Next: 1
Next: 3
Next: 6
Next: 10
Next: 15
Sequence complete.

```

- Javadoc: `scan(Func2)`



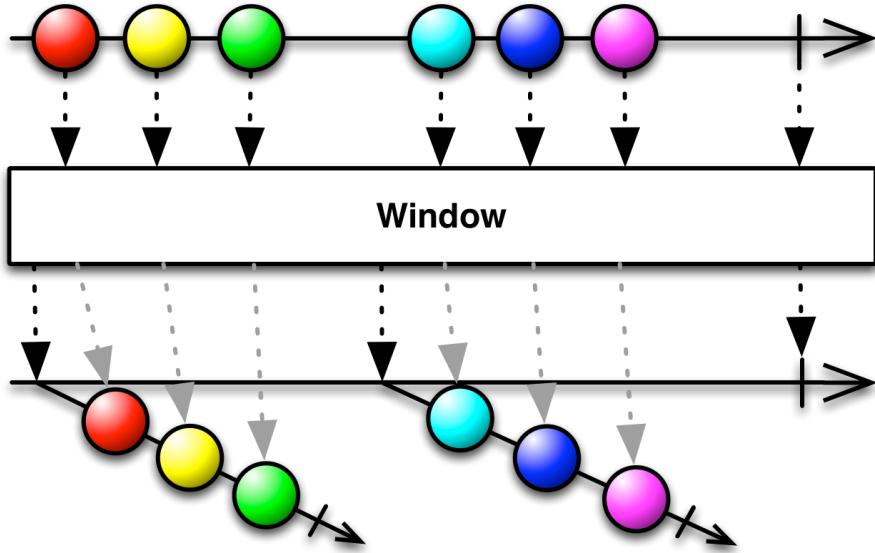
有一个 `scan` 操作符的变体，你可以传递一个种子值给累加器函数的第一次调用（`Observable` 发射的第一项数据）。如果你使用这个版本，`scan` 将发射种子值作为自己的第一项数据。注意：传递 `null` 作为种子值与不传递是不同的，`null` 种子值是合法的。

- Javadoc: `scan(R,Func2)`

这个操作符默认不在任何特定的调度器上执行。

Window

定期将来自原始 `Observable` 的数据分解为一个 `Observable` 窗口，发射这些窗口，而不是每次发射一项数据

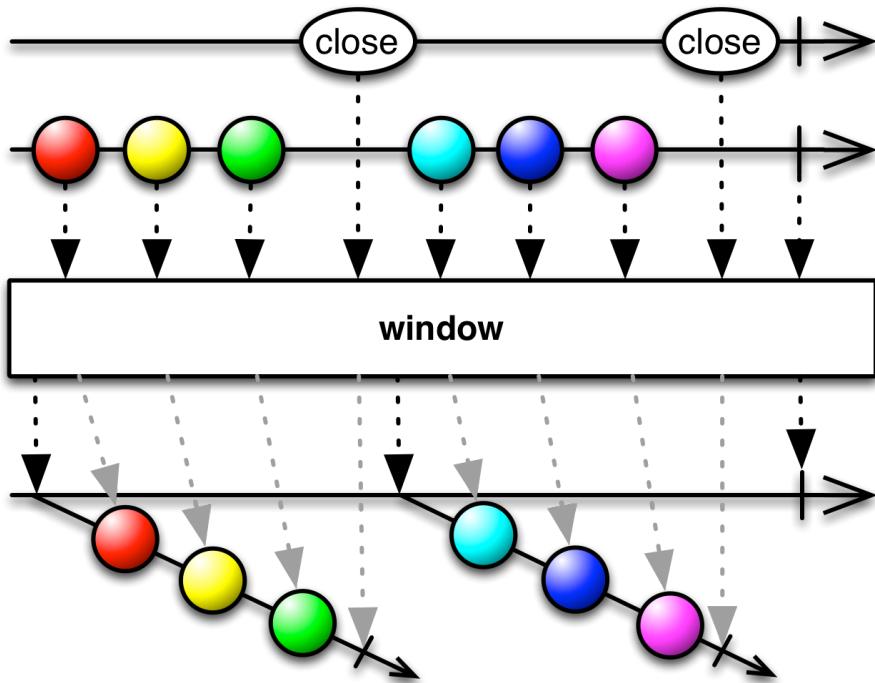


`window` 和 `Buffer` 类似，但不是发射来自原始Observable的数据包，它发射的是 Observables，这些Observables中的每一个都发射原始Observable数据的一个子集，最后发射一个 `onCompleted` 通知。

和 `Buffer` 一样，`window` 有很多变体，每一种都以自己的方式将原始Observable 分解为多个作为结果的Observable，每一个都包含一个映射原始数据的 `window`。用 `window` 操作符的术语描述就是，当一个窗口打开(`when a window "opens"`)意味着一个新的Observable已经发射(产生)了，而且这个Observable 开始发射来自原始Observable的数据；当一个窗口关闭(`when a window "closes"`)意味着发射(产生)的Observable停止发射原始Observable的数据，并且发射终止通知 `onCompleted` 给它的观察者们。

在RxJava中有许多种 `window` 操作符的变体。

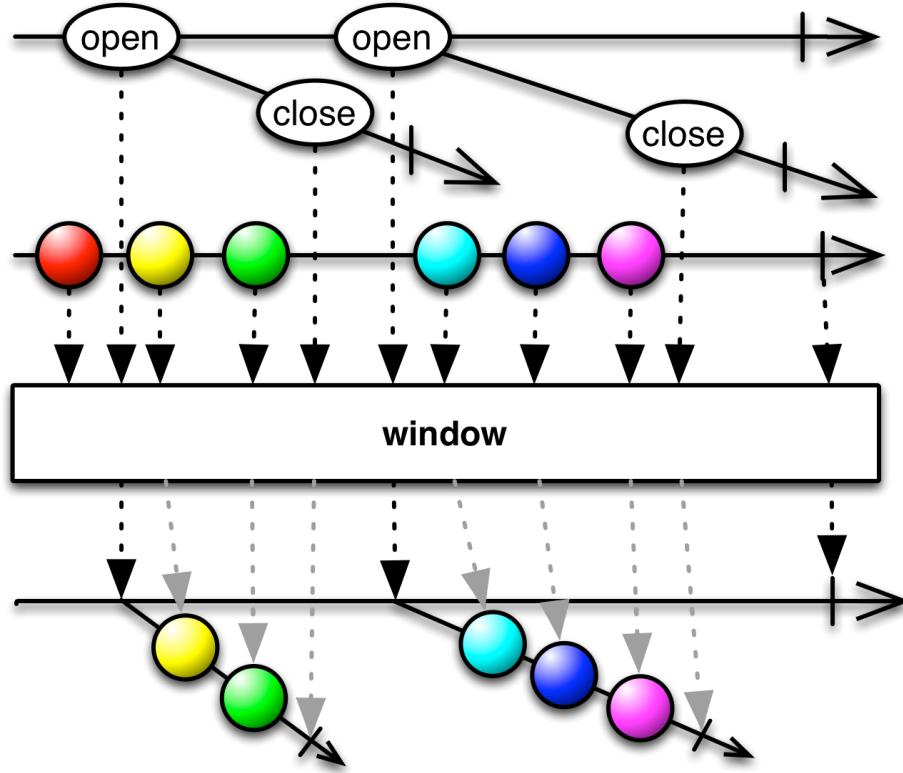
window(closingSelector)



`window`的这个变体会立即打开它的第一个窗口。每当它观察到`closingSelector`返回的Observable发射了一个对象时，它就关闭当前打开的窗口并立即打开一个新窗口。用这个方法，这种`window`变体发射一系列不重叠的窗口，这些窗口的数据集合与原始Observable发射的数据是一一对应的。

- Javadoc: [window\(Func0\)](#)

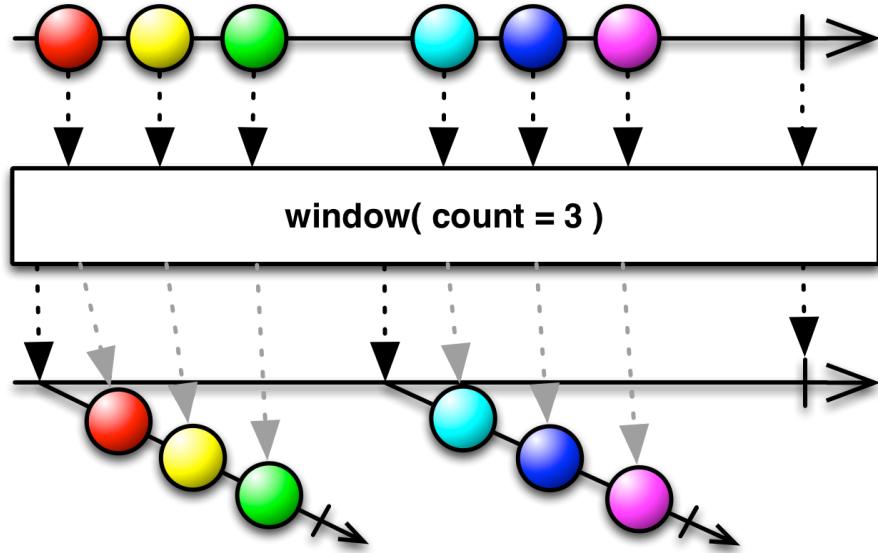
window(windowOpenings, closingSelector)



无论何时，只要`window`观察到`windowOpenings`这个Observable发射了一个`Opening`对象，它就打开一个窗口，并且同时调用`closingSelector`生成一个与那个窗口关联的关闭(closing)Observable。当这个关闭(closing)Observable发射了一个对象时，`window`操作符就会关闭那个窗口。对这个变体来说，由于当前窗口的关闭和新窗口的打开是由单独的Observable管理的，它创建的窗口可能会存在重叠（重复某些来自原始Observable的数据）或间隙（丢弃某些来自原始Observable的数据）。

- Javadoc: [window\(Observable, Func1\)](#)

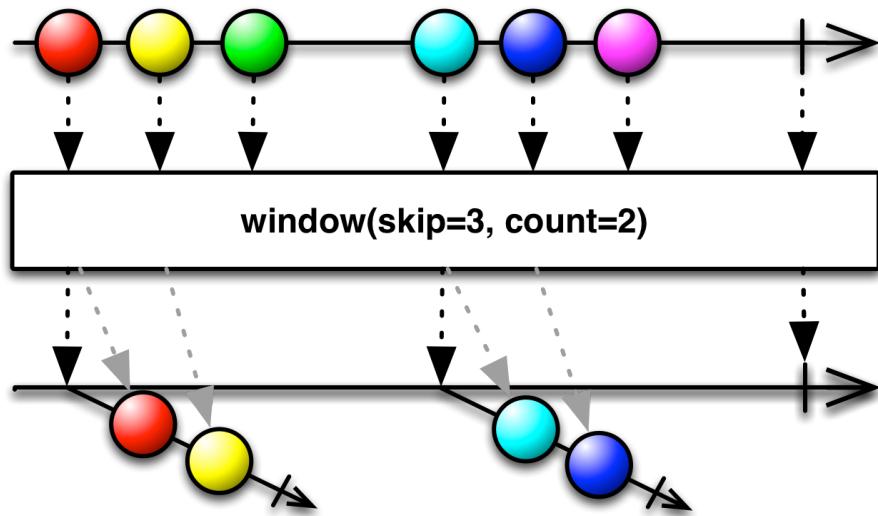
window(count)



这个 `window` 的变体立即打开它的第一个窗口。每当当前窗口发射了 `count` 项数据，它就关闭当前窗口并打开一个新窗口。如果从原始 `Observable` 收到了 `onError` 或 `onCompleted` 通知它也会关闭当前窗口。这种 `window` 变体发射一系列不重叠的窗口，这些窗口的数据集合与原始 `Observable` 发射的数据是一一对应的。

- Javadoc: [window\(int\)](#)

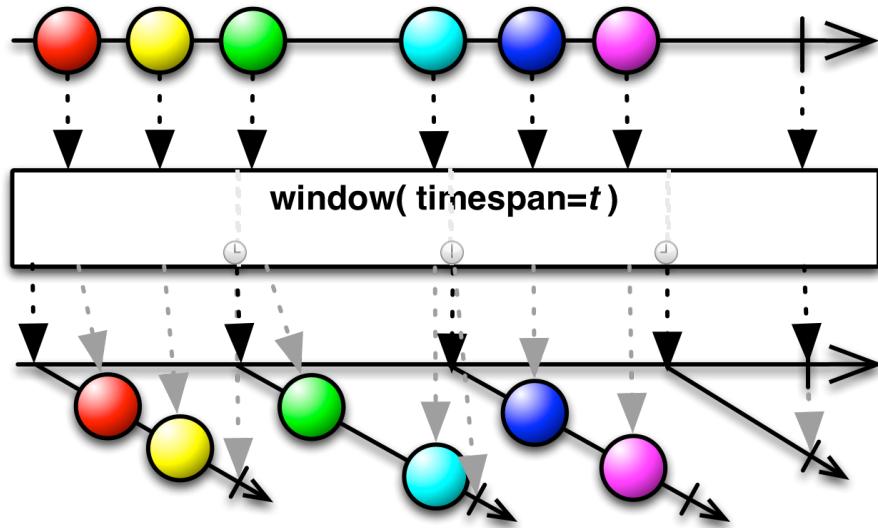
`window(count, skip)`



这个 `window` 的变体立即打开它的第一个窗口。原始 `Observable` 每发射 `skip` 项数据它就打开一个新窗口（例如，如果 `skip` 等于 3，每到第三项数据，它会打开一个新窗口）。每当当前窗口发射了 `count` 项数据，它就关闭当前窗口并打开一个新窗口。如果从原始 `Observable` 收到了 `onError` 或 `onCompleted` 通知它也会关闭当前窗口。如果 `skip=count`，它的行为与 `window(source, count)` 相同；如果 `skip < count`，窗口可会有 `count - skip` 个重叠的数据；如果 `skip > count`，在两个窗口之间会有 `skip - count` 项数据被丢弃。

- Javadoc: [window\(int,int\)](#)

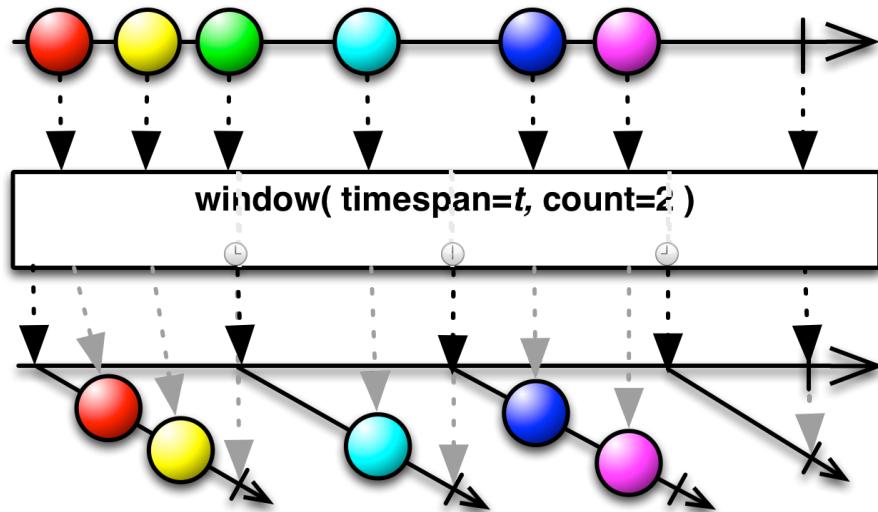
`window(timespan, unit[, scheduler])`



这个 `window` 的变体立即打开它的第一个窗口。每当过了 `timespan` 这么长的时间它就关闭当前窗口并打开一个新窗口（时间单位是 `unit`，可选在调度器 `scheduler` 上执行）。如果从原始 `Observable` 收到了 `onError` 或 `onCompleted` 通知它也会关闭当前窗口。这种 `window` 变体发射一系列不重叠的窗口，这些窗口的数据集合与原始 `Observable` 发射的数据也是一一对应的。

- Javadoc: `window(long, TimeUnit)`
- Javadoc: `window(long, TimeUnit, Scheduler)`

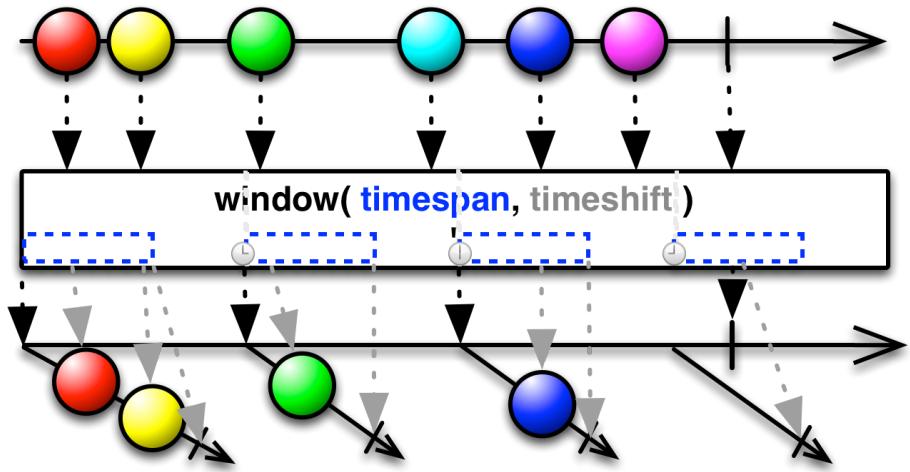
`window(timespan, unit, count[, scheduler])`



这个 `window` 的变体立即打开它的第一个窗口。这个变体是 `window(count)` 和 `window(timespan, unit[, scheduler])` 的结合，每当过了 `timespan` 的时长或者当前窗口收到了 `count` 项数据，它就关闭当前窗口并打开另一个。如果从原始 `Observable` 收到了 `onError` 或 `onCompleted` 通知它也会关闭当前窗口。这种 `window` 变体发射一系列不重叠的窗口，这些窗口的数据集合与原始 `Observable` 发射的数据也是一一对应的。

- Javadoc: `window(long, TimeUnit, int)`
- Javadoc: `window(long, TimeUnit, int, Scheduler)`

`window(timespan, timeshift, unit[, scheduler])`



`buffer(timespan, timeshift, unit)` 在每一个 `timeshift` 时期内都创建一个 `List`, 然后用原始 Observable 发射的每一项数据填充这个列表 (在把这个 `List` 当做自己的数据发射前, 从创建时开始, 直到过了 `timespan` 这么长的时间)。如果 `timespan` 长于 `timeshift`, 它发射的数据包将会重叠, 因此可能包含重复的数据项。

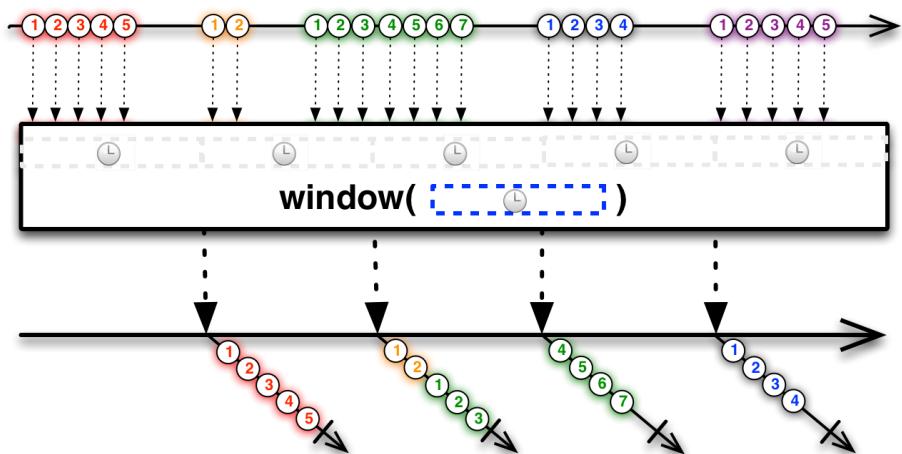
这个 `window` 的变体立即打开它的第一个窗口。随后每当过了 `timeshift` 的时长就打开一个新窗口 (时间单位是 `unit`, 可选在调度器 `scheduler` 上执行), 当窗口打开的时长达到 `timespan`, 它就关闭当前打开的窗口。如果从原始 Observable 收到了 `onError` 或 `onCompleted` 通知它也会关闭当前窗口。窗口的数据可能重叠也可能有间隙, 取决于你设置的 `timeshift` 和 `timespan` 的值。

这个变体的 `window` 默认在 `computation` 调度器上执行它的定时器。

- Javadoc: `window(long,long,TimeUnit)`
- Javadoc: `window(long,long,TimeUnit,Scheduler)`

window-backpressure

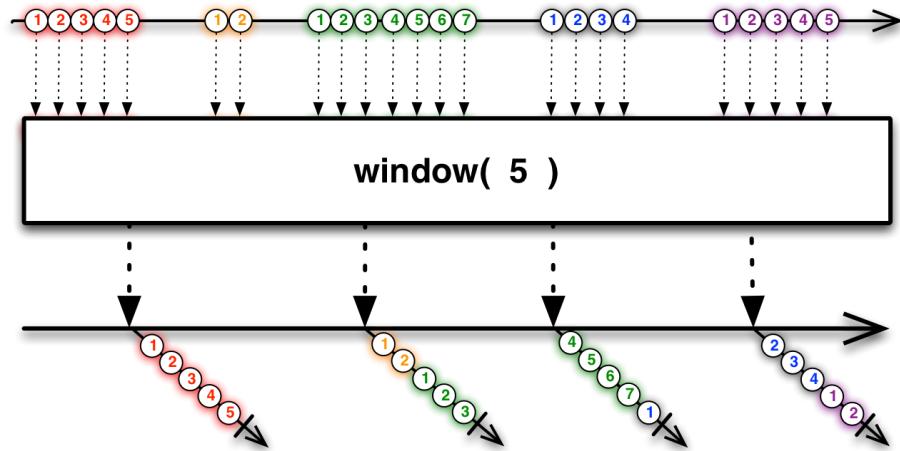
你可以使用 `window` 操作符实现反压 `backpressure` (意思是, 处理这样一个 Observable: 它产生数据的数据可能比它的观察者消费数据的数据快)。



`Window` 操作符可以将大量的数据序列缩减为较少的数据窗口序列, 让它们更容易处理。例如, 你可以按固定的时间间隔, 定期关闭和发射来自一个爆发性 Observable 的数据窗口。

示例代码

```
Observable<Observable<Integer>> burstyWindowed =  
    bursty.window(500, TimeUnit.MILLISECONDS);
```



你还可以选择每当收到爆发性Observable的N项数据时发射一个新的数据窗口。

示例代码

```
Observable<Observable<Integer>> burstyWindowed =  
    bursty.window(5);
```

过滤操作

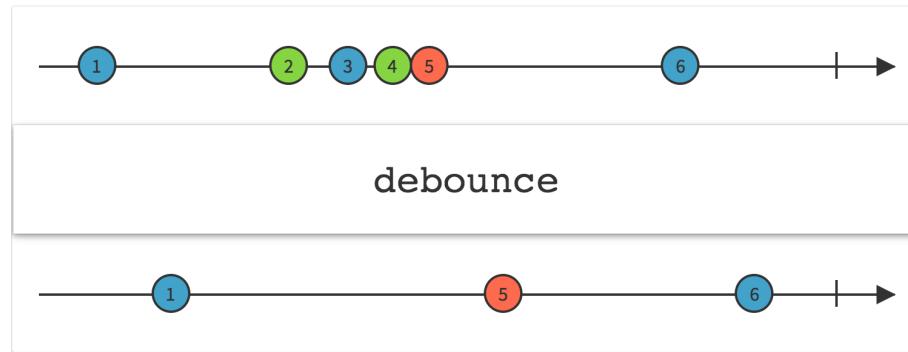
这个页面展示的操作符可用于过滤和选择Observable发射的数据序列。

- **filter()** – 过滤数据
- **takeLast()** – 只发射最后的N项数据
- **last()** – 只发射最后一项数据
- **lastOrDefault()** – 只发射最后一项数据，如果Observable为空就发射默认值
- **takeLastBuffer()** – 将最后的N项数据当做单个数据发射
- **skip()** – 跳过开始的N项数据
- **skipLast()** – 跳过最后的N项数据
- **take()** – 只发射开始的N项数据
- **first() and takeFirst()** – 只发射第一项数据，或者满足某种条件的第一项数据
- **firstOrDefault()** – 只发射第一项数据，如果Observable为空就发射默认值
- **elementAt()** – 发射第N项数据
- **elementAtOrDefault()** – 发射第N项数据，如果Observable数据少于N项就发射默认值
- **sample() or throttleLast()** – 定期发射Observable最近的数据
- **throttleFirst()** – 定期发射Observable发射的第一项数据
- **throttleWithTimeout() or debounce()** – 只有当Observable在指定的时间后还没有发射数据时，才发射一个数据

- **timeout()** – 如果在一个指定的时间段后还没发射数据，就发射一个异常
- **distinct()** – 过滤掉重复数据
- **distinctUntilChanged()** – 过滤掉连续重复的数据
- **ofType()** – 只发射指定类型的数据
- **ignoreElements()** – 丢弃所有的正常数据，只发射错误或完成通知

Debounce

仅在过了一段指定的时间还没发射数据时才发射一个数据

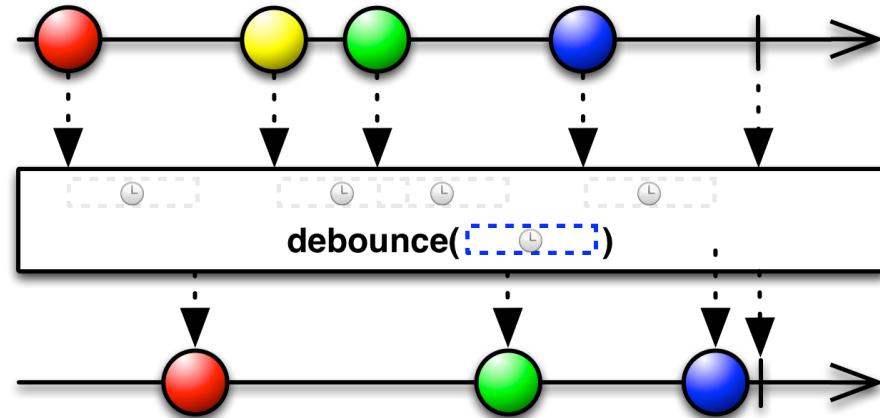


`Debounce`操作符会过滤掉发射速率过快的数据项。

RxJava将这个操作符实现为`throttleWithTimeout`和`debounce`。

注意：这个操作符会接着最后一项数据发射原始Observable的`onCompleted`通知，即使这个通知发生在你指定的时间窗口内（从最后一项数据的发射算起）。也就是说，`onCompleted`通知不会触发限流。

throttleWithTimeout



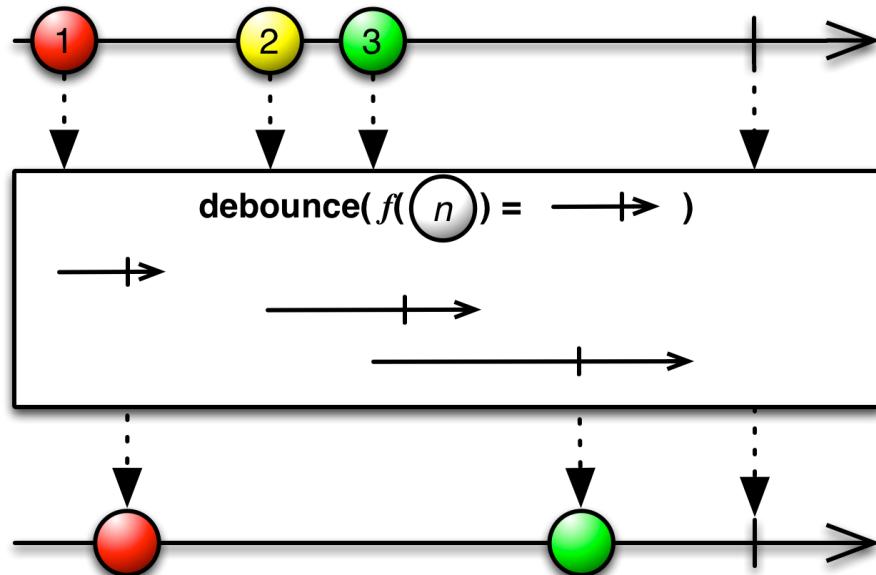
`throttleWithTimeout/debounce`的一个变体根据你指定的时间间隔进行限流，时间单位通过`TimeUnit`参数指定。

这种操作符默认在`computation`调度器上执行，但是你可以通过第三个参数指定。

- Javadoc: `throttleWithTimeout(long, TimeUnit)` and `debounce(long, TimeUnit)`

- Javadoc: `throttleWithTimeout(long, TimeUnit, Scheduler)` and `debounce(long, TimeUnit, Scheduler)`

debounce

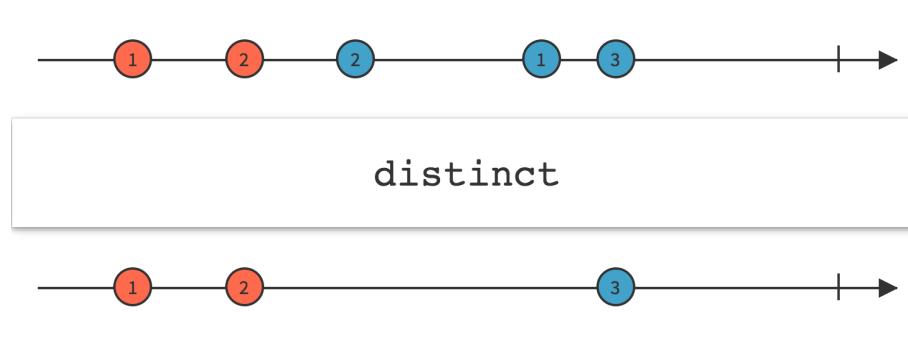


`debounce` 操作符的一个变体通过对原始Observable的每一项应用一个函数进行限流，这个函数返回一个Observable。如果原始Observable在这个新生成的 Observable终止之前发射了另一个数据，`debounce`会抑制(suppress)这个数据项。

`debounce` 的这个变体默认不在任何特定的调度器上执行。

Distinct

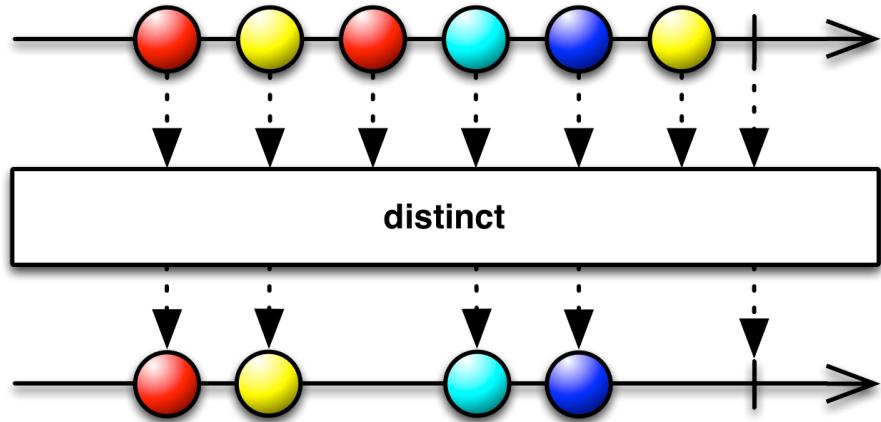
抑制（过滤掉）重复的数据项



`distinct` 的过滤规则是：只允许还没有发射过的数据项通过。

在某些实现中，有一些变体允许你调整判定两个数据不同(`distinct`)的标准。还有一些实现只比较一项数据和它的直接前驱，因此只会从序列中过滤掉连续重复的数据。

`distinct()`



RxJava将这个操作符实现为`distinct`函数。

示例代码

```
Observable.just(1, 2, 1, 1, 2, 3)
    .distinct()
    .subscribe(new Subscriber<Integer>() {
        @Override
        public void onNext(Integer item) {
            System.out.println("Next: " + item);
        }

        @Override
        public void onError(Throwable error) {
            System.err.println("Error: " +
                error.getMessage());
        }

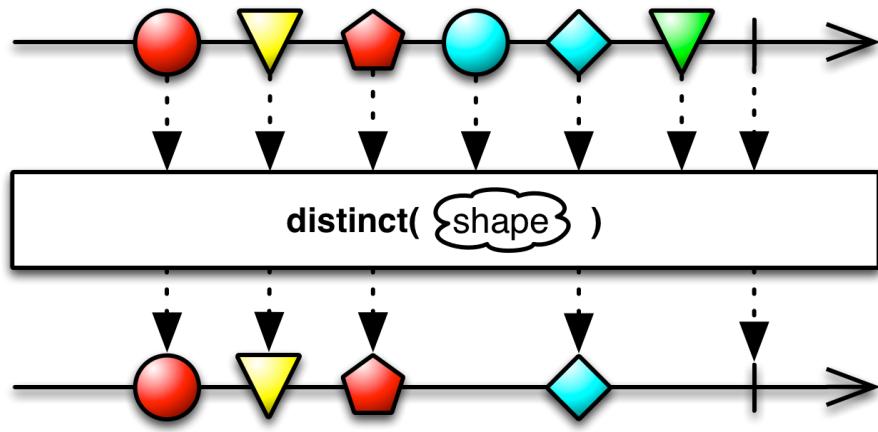
        @Override
        public void onCompleted() {
            System.out.println("Sequence complete.");
        }
    });
}
```

输出

```
Next: 1
Next: 2
Next: 3
Sequence complete.
```

- Javadoc: `distinct()`

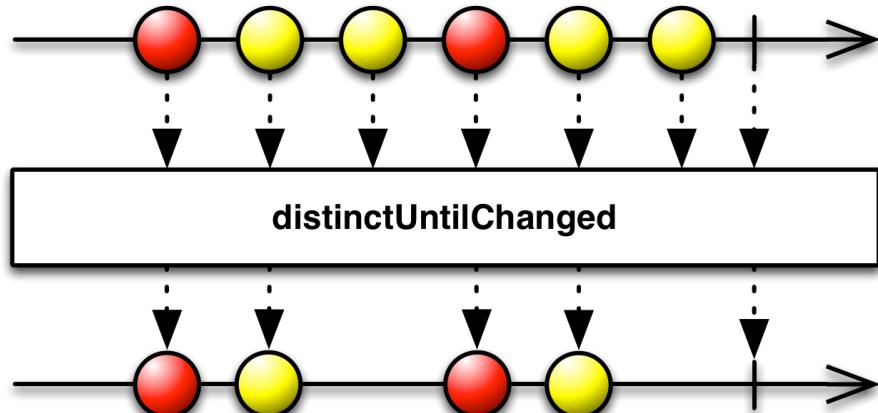
`distinct(Func1)`



这个操作符有一个变体接受一个函数。这个函数根据原始Observable发射的数据项产生一个Key，然后，比较这些Key而不是数据本身，来判定两个数据是否是不同的。

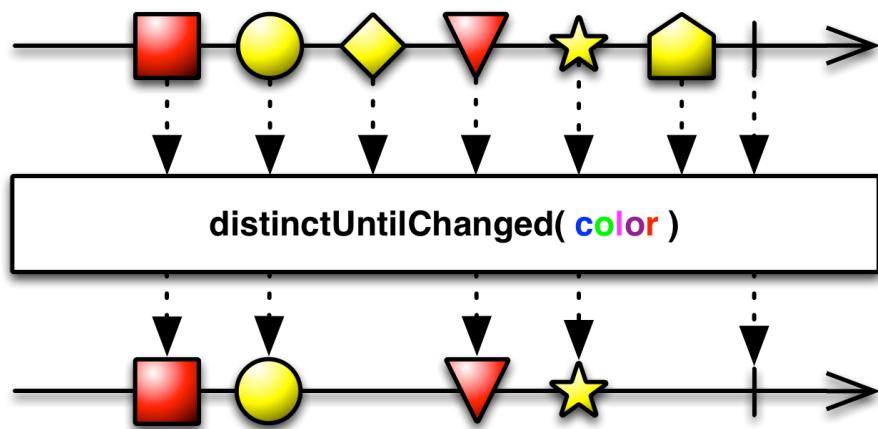
- Javadoc: [distinct\(Func1\)](#)

distinctUntilChanged



RxJava还是实现了一个 `distinctUntilChanged` 操作符。它只判定一个数据和它的直接前驱是否是不同的。

distinctUntilChanged(Func1)



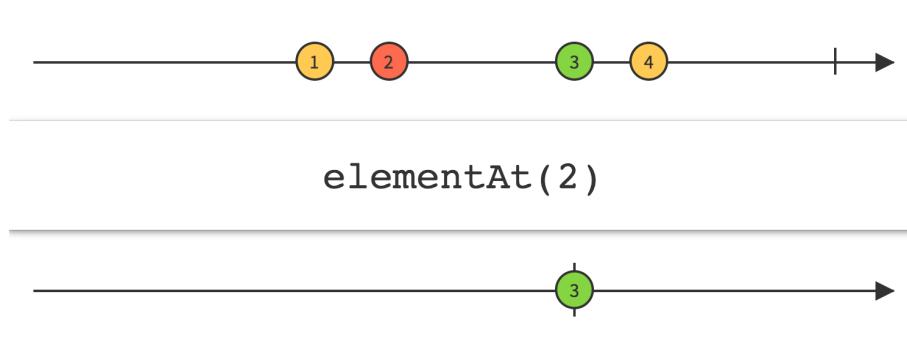
和 `distinct(Func1)` 一样，根据一个函数产生的Key判定两个相邻的数据项是不是不同的。

- Javadoc: `distinctUntilChanged(Func1)`

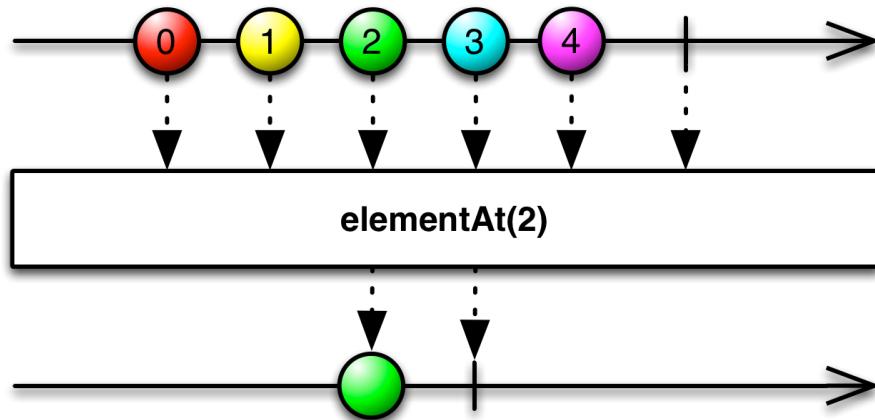
`distinct` 和 `distinctUntilChanged` 默认不在任何特定的调度器上执行。

ElementAt

只发射第N项数据



`ElementAt` 操作符获取原始 Observable 发射的数据序列指定索引位置的数据项，然后当做自己的唯一数据发射。

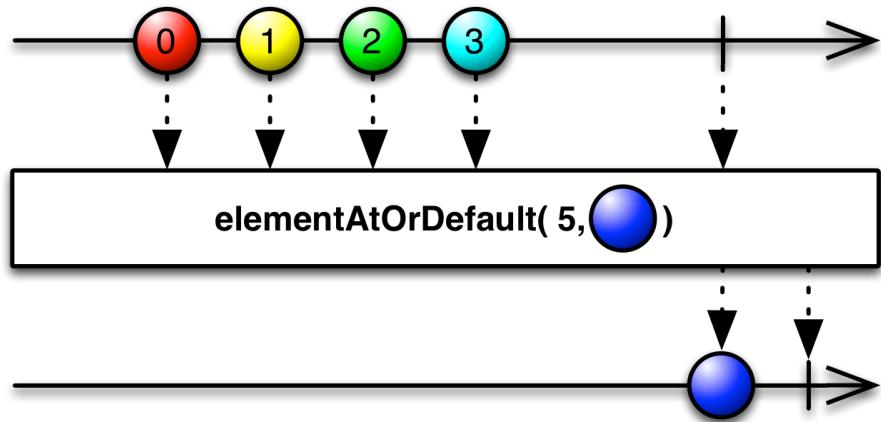


RxJava 将这个操作符实现为 `elementAt`，给它传递一个基于 0 的索引值，它会发射原始 Observable 数据序列对应索引位置的值，如果你传递给 `elementAt` 的值为 5，那么它会发射第六项的数据。

如果你传递的是一个负数，或者原始 Observable 的数据项数小于 `index+1`，将会抛出一个 `IndexOutOfBoundsException` 异常。

- Javadoc: `elementAt(int)`

elementAtOrDefault



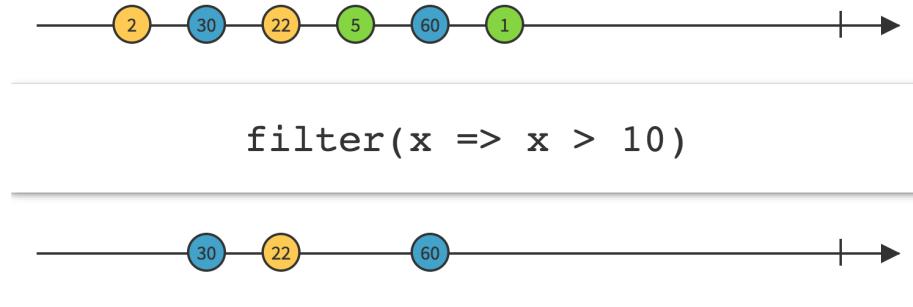
RxJava还实现了 `elementAtOrDefault` 操作符。与 `elementAt` 的区别是，如果索引值大于数据项数，它会发射一个默认值（通过额外的参数指定），而不是抛出异常。但是如果你传递一个负数索引值，它仍然会抛出一个 `IndexOutOfBoundsException` 异常。

- Javadoc: `elementAtOrDefault(int,T)`

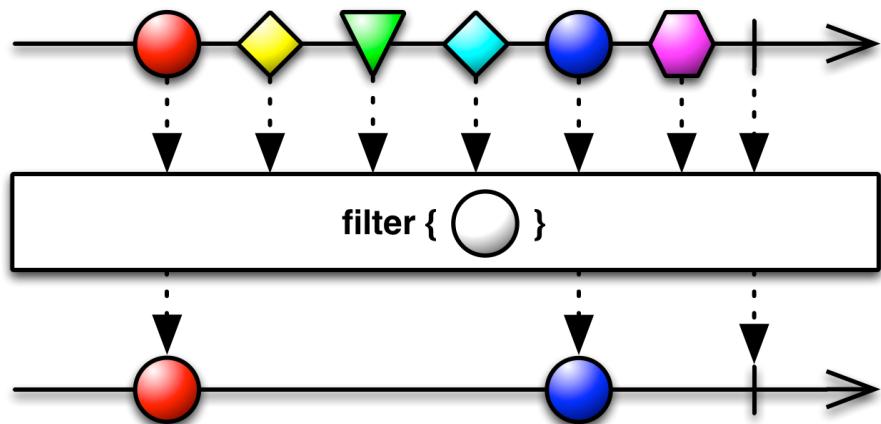
`elementAt` 和 `elementAtOrDefault` 默认不在任何特定的调度器上执行。

Filter

只发射通过了谓词测试的数据项



`Filter` 操作符使用你指定的一个谓词函数测试数据项，只有通过测试的数据才会被发射。



RxJava将这个操作符实现为 `filter` 函数。

示例代码

```
observable.just(1, 2, 3, 4, 5)
    .filter(new Func1<Integer, Boolean>() {
        @Override
        public Boolean call(Integer item) {
            return( item < 4 );
        }
    }).subscribe(new Subscriber<Integer>() {
        @Override
        public void onNext(Integer item) {
            System.out.println("Next: " + item);
        }

        @Override
        public void onError(Throwable error) {
            System.err.println("Error: " +
error.getMessage());
        }

        @Override
        public void onCompleted() {
            System.out.println("Sequence complete.");
        }
    });
}
```

输出

```
Next: 1
Next: 2
Next: 3
Sequence complete.
```

`filter`默认不在任何特定的调度器上执行。

- Javadoc: [filter\(Func1\)](#)

ofType

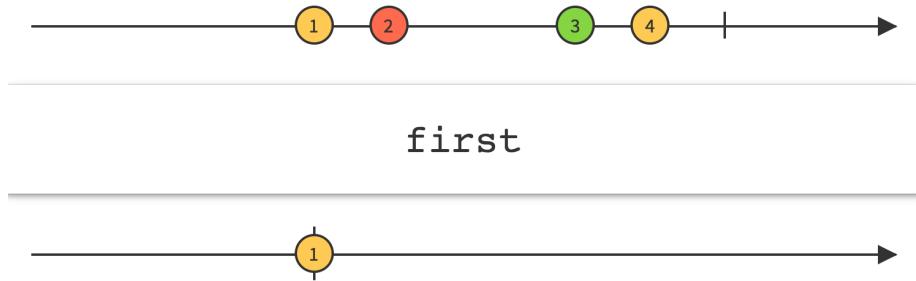
`ofType`是`filter`操作符的一个特殊形式。它过滤一个Observable只返回指定类型的数据。

`ofType`默认不在任何特定的调度器上指定。

- Javadoc: [ofType\(Class\)](#)

First

只发射第一项（或者满足某个条件的第一项）数据



如果你只对Observable发射的第一项数据，或者满足某个条件的第一项数据感兴趣，你可以使用`First`操作符。

在某些实现中，`First`没有实现为一个返回Observable的过滤操作符，而是实现为一个在当时就发射原始Observable指定数据项的阻塞函数。在这些实现中，如果你想要的是一个过滤操作符，最好使用`Take(1)`或者`ElementAt(0)`。

在一些实现中还有一个`single`操作符。它的行为与`First`类似，但为了确保只发射单个值，它会等待原始Observable终止（否则，不是发射那个值，而是以一个错误通知终止）。你可以使用它从原始Observable获取第一项数据，而且也确保只发射一项数据。

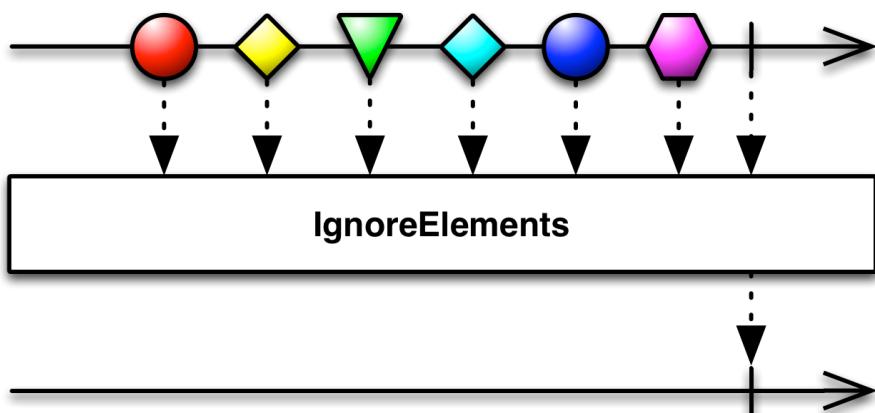
在RxJava中，这个操作符被实现为`first`，`firstOrDefault`和`takeFirst`。

可能容易混淆，`BlockingObservable`也有名叫`first`和`firstOrDefault`的操作符，它们会阻塞并返回值，不是立即返回一个Observable。

还有几个其它的操作符执行类似的功能。

IgnoreElements

不发射任何数据，只发射Observable的终止通知



`IgnoreElements`操作符抑制原始Observable发射的所有数据，只允许它的终止通知（`onError`或`onCompleted`）通过。

如果你不关心一个Observable发射的数据，但是希望在它完成时或遇到错误终止时收到通知，你可以对Observable使用`ignoreElements`操作符，它会确保永远不会调用观察者的`onNext()`方法。

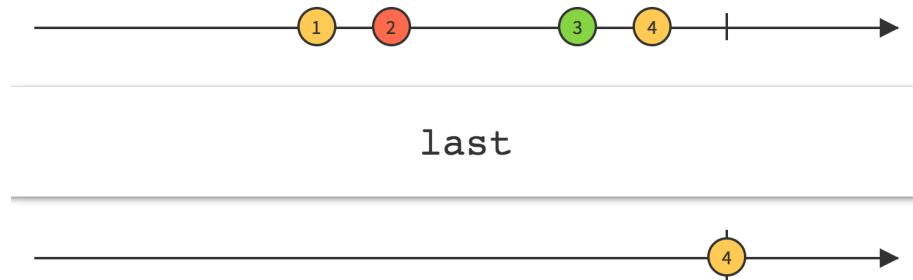
RxJava将这个操作符实现为`ignoreElements`。

- Javadoc: `ignoreElements()`

`ignoreElements`默认不在任何特定的调度器上执行。

Last

只发射最后一项（或者满足某个条件的最后一项）数据



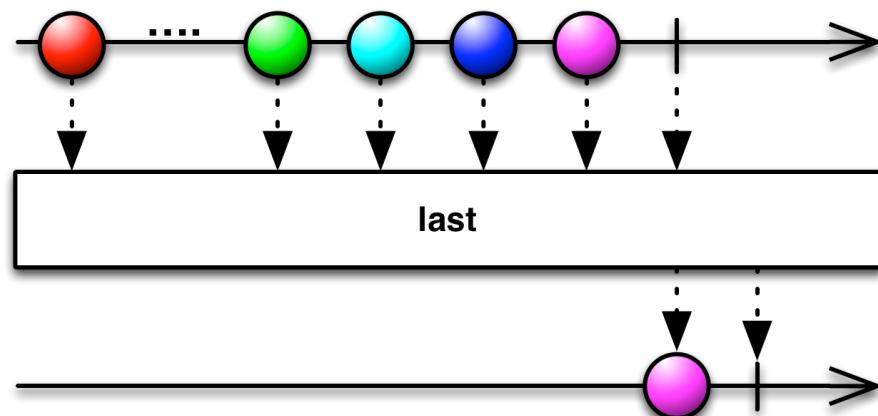
如果你只对Observable发射的最后一项数据，或者满足某个条件的最后一项数据感兴趣，你可以使用`Last`操作符。

在某些实现中，`Last`没有实现为一个返回Observable的过滤操作符，而是实现为一个在当时就发射原始Observable指定数据项的阻塞函数。在这些实现中，如果你想要的是一个过滤操作符，最好使用`TakeLast(1)`。

在RxJava中的实现是`Last`和`LastOrDefault`。

可能容易混淆，`BlockingObservable`也有名叫`Last`和`LastOrDefault`的操作符，它们会阻塞并返回值，不是立即返回一个Observable。

过滤操作符



只发射最后一项数据，使用没有参数的`Last`操作符。

示例代码

```
Observable.just(1, 2, 3)
    .last()
```

```

        .subscribe(new Subscriber<Integer>() {
            @Override
            public void onNext(Integer item) {
                System.out.println("Next: " + item);
            }

            @Override
            public void onError(Throwable error) {
                System.err.println("Error: " +
error.getMessage());
            }

            @Override
            public void onCompleted() {
                System.out.println("Sequence complete.");
            }
        });
    }
}

```

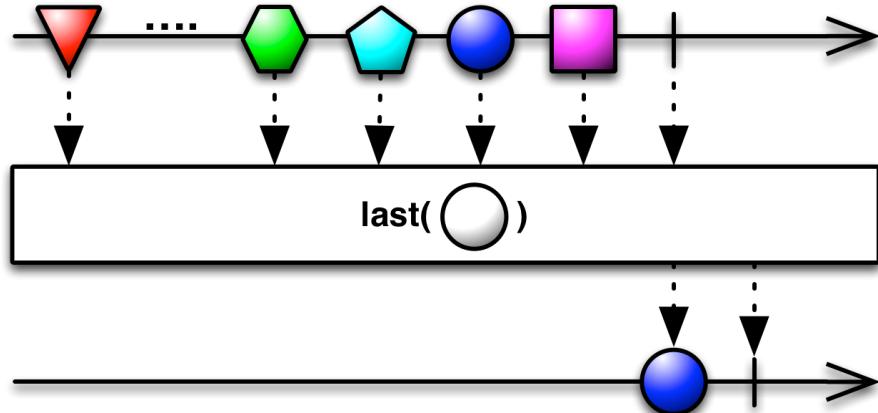
输出

```

Next: 3
Sequence complete.

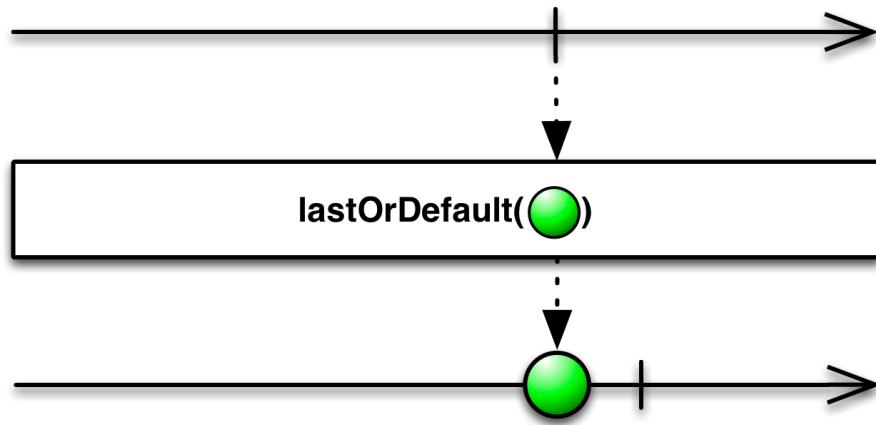
```

- Javadoc: `last()`



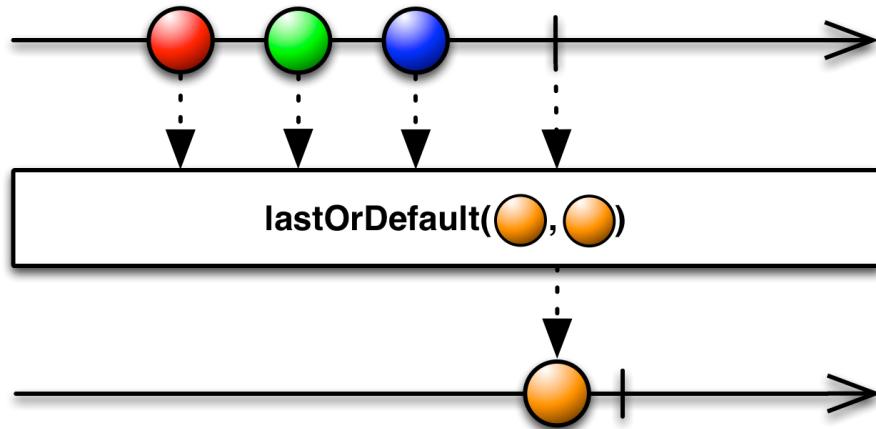
这个版本的 `last` 也是接受一个谓词函数，返回一个发射原始 Observable 中满足条件的最后一项数据的 Observable。

- Javadoc: `last(Func1)`



`lastOrDefault`与`last`类似，不同的是，如果原始Observable没有发射任何值，它发射你指定的默认值。

- Javadoc: [lastOrDefault\(T\)](#)



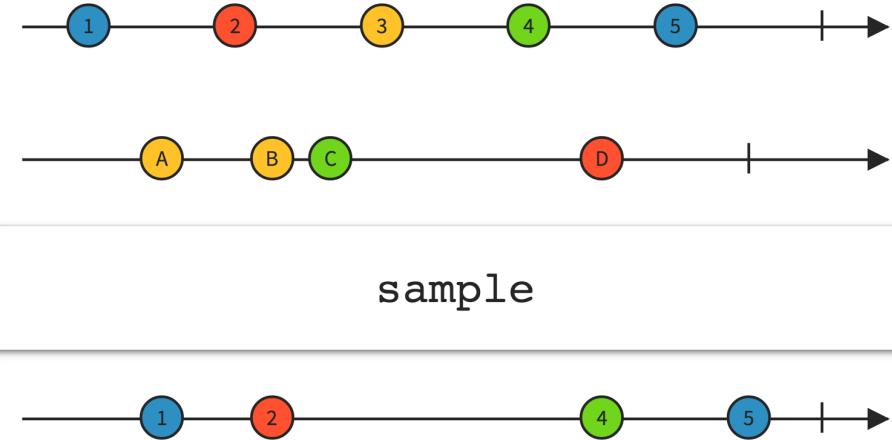
这个版本的`lastOrDefault`可以接受一个谓词函数，如果有数据满足条件，返回的Observable就发射原始Observable满足条件的最后一项数据，否则发射默认值。

- Javadoc: [lastOrDefault\(T\)](#)

`last`和`lastOrDefault`默认不在任何特定的调度器上执行。

Sample

定期发射Observable最近发射的数据项

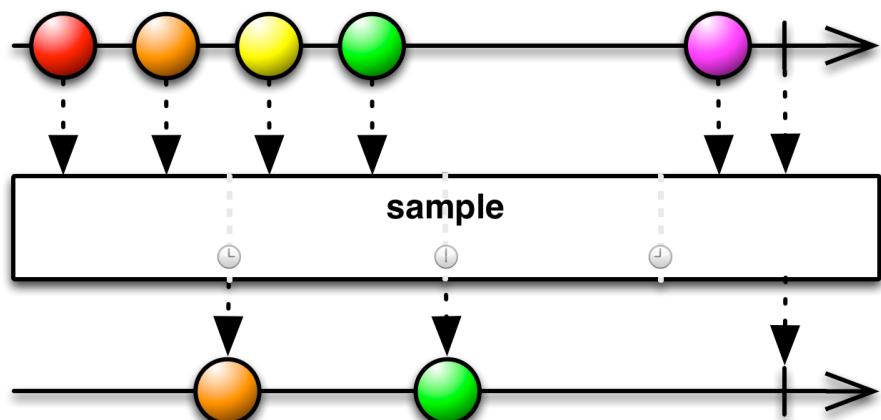


`sample` 操作符定时查看一个Observable，然后发射自上次采样以来它最近发射的数据。

在某些实现中，有一个`ThrottleFirst`操作符的功能类似，但不是发射采样期间的最近的数据，而是发射在那段时间内的第一项数据。

RxJava将这个操作符实现为`sample`和`throttleLast`。

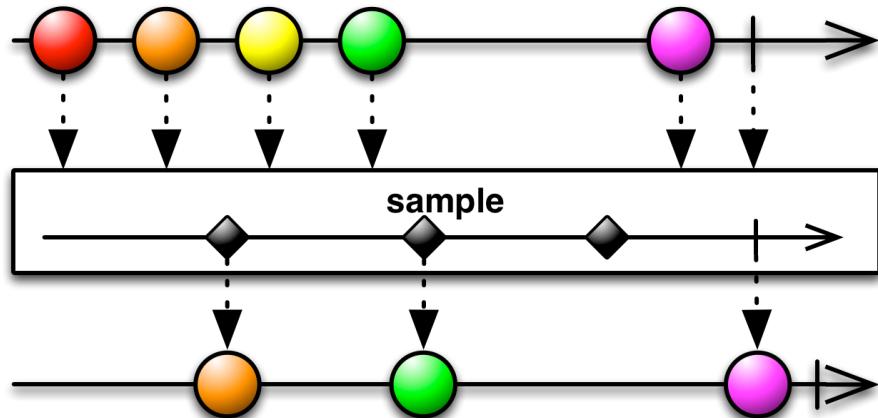
注意：如果自上次采样以来，原始Observable没有发射任何数据，这个操作返回的Observable在那段时间内也不会发射任何数据。



`sample`(别名`throttleLast`)的一个变体按照你参数中指定的时间间隔定时采样(`TimeUnit`指定时间单位)。

`sample`的这个变体默认在`computation`调度器上执行，但是你可以使用第三个参数指定其它的调度器。

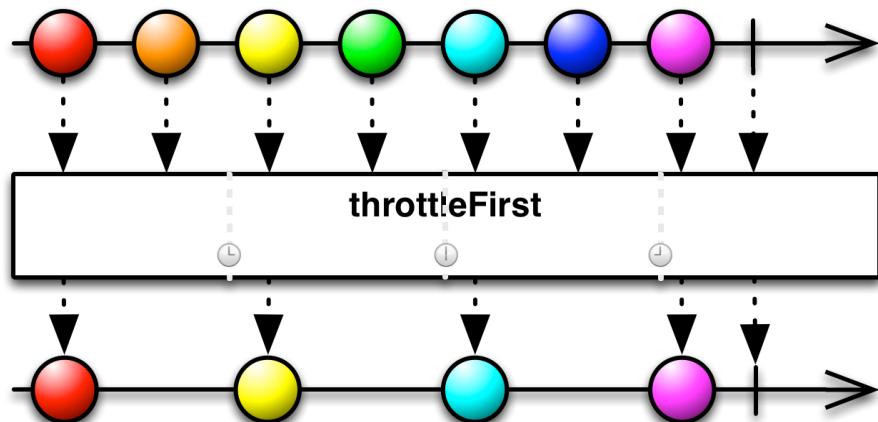
- Javadoc: `sample(long,TimeUnit)`和`throttleLast(long,TimeUnit)`
- Javadoc: `sample(long,TimeUnit,Scheduler)`和`throttleLast(long,TimeUnit,Scheduler)`



`sample` 的这个变体每当第二个Observable发射一个数据（或者当它终止）时就对原始Observable进行采样。第二个Observable通过参数传递给 `sample`。

`sample` 的这个变体默认不在任何特定的调度器上执行。

- Javadoc: [sample\(Observable\)](#)



`throttleFirst` 与 `throttleLast/sample` 不同，在每个采样周期内，它总是发射原始Observable的第一项数据，而不是最近的一项。

`throttleFirst` 操作符默认在 `computation` 调度器上执行，但是你可以使用第三个参数指定其它的调度器。

- Javadoc: [throttleFirst\(long,TimeUnit\)](#)
- Javadoc: [throttleFirst\(long,TimeUnit,Scheduler\)](#)

Skip

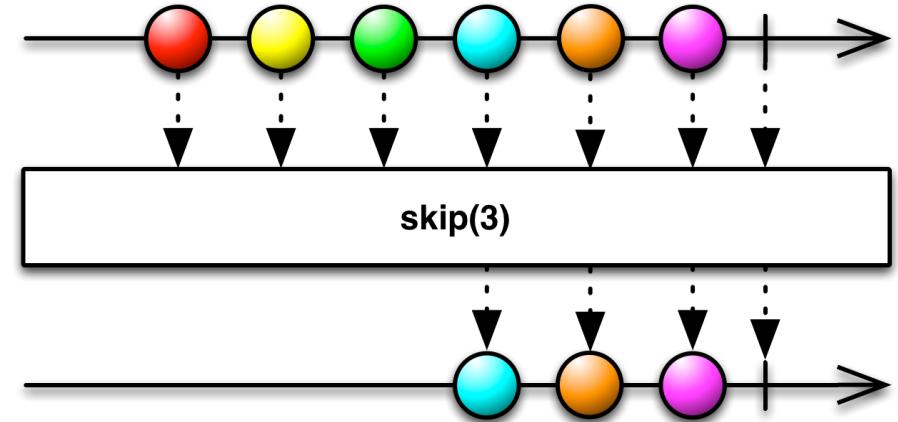
抑制Observable发射的前N项数据



`skip(2)`

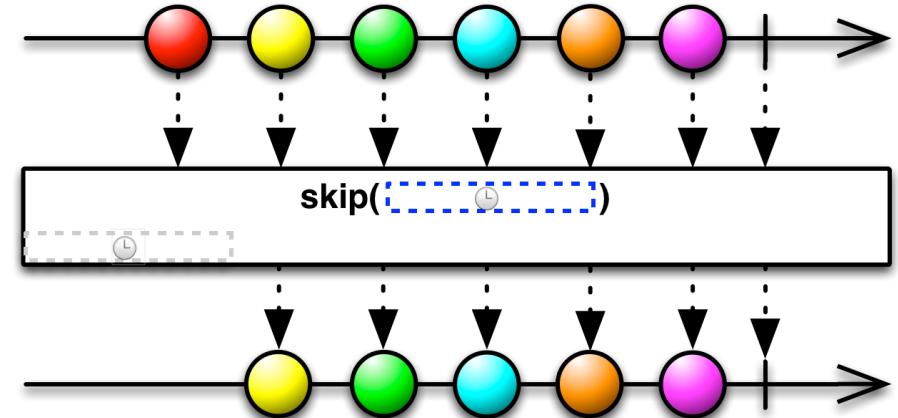


使用 `skip` 操作符，你可以忽略 Observable' 发射的前 N 项数据，只保留之后的数据。



RxJava 中这个操作符叫 `skip`。`skip` 的这个变体默认不在任何特定的调度器上执行。

- Javadoc: [skip\(int\)](#)



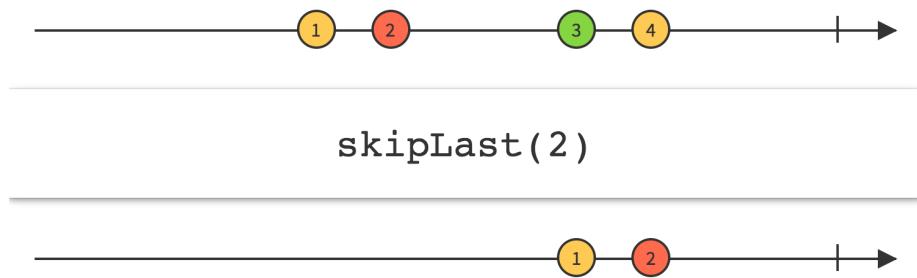
`skip` 的这个变体接受一个时长而不是数量参数。它会丢弃原始 Observable 开始的那段时间发射的数据，时长和时间单位通过参数指定。

`skip` 的这个变体默认在 `computation` 调度器上执行，但是你可以使用第三个参数指定其它的调度器。

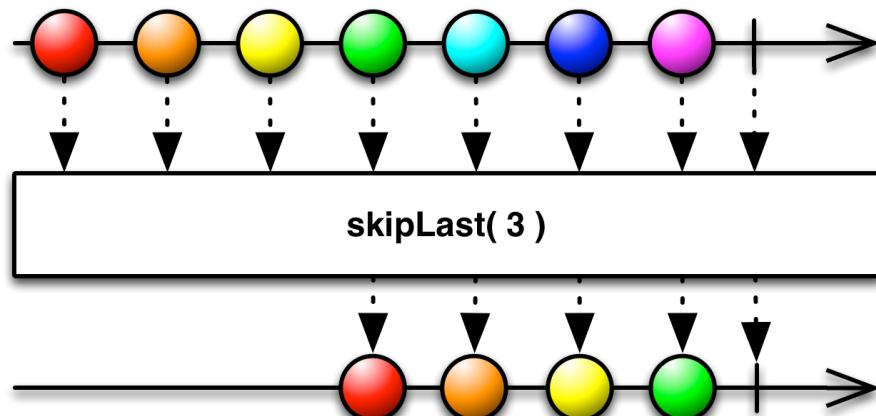
- Javadoc: [skip\(long, TimeUnit\)](#)
- Javadoc: [skip\(long, TimeUnit, Scheduler\)](#)

SkipLast

抑制Observable发射的后N项数据



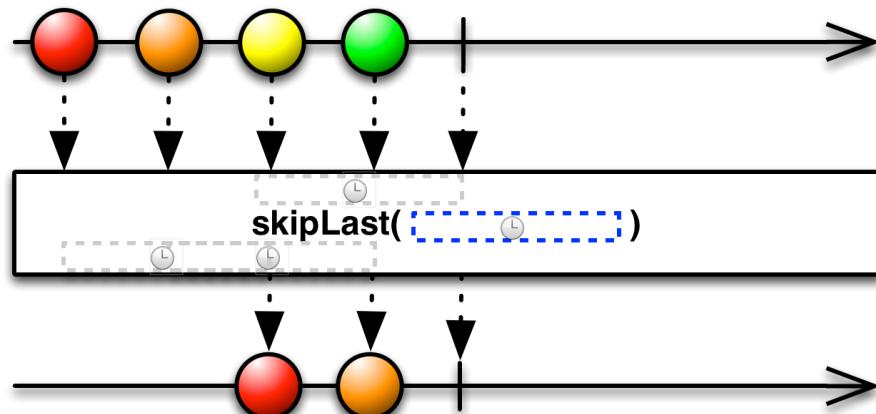
使用 `skipLast` 操作符修改原始 Observable，你可以忽略 Observable' 发射的后 N 项数据，只保留前面的数据。



使用 `skipLast` 操作符，你可以忽略原始 Observable' 发射的后 N 项数据，只保留之前的数据。注意：这个机制是这样实现的：延迟原始 Observable' 发射的任何数据项，直到它发射了 N 项数据。

`skipLast` 的这个变体默认不在任何特定的调度器上执行。

- Javadoc: [skipLast\(int\)](#)



还有一个 `skipLast` 变体接受一个时长而不是数量参数。它会丢弃在原始 Observable 的生命周期内最后一段时间内发射的数据。时长和时间单位通过参数指定。

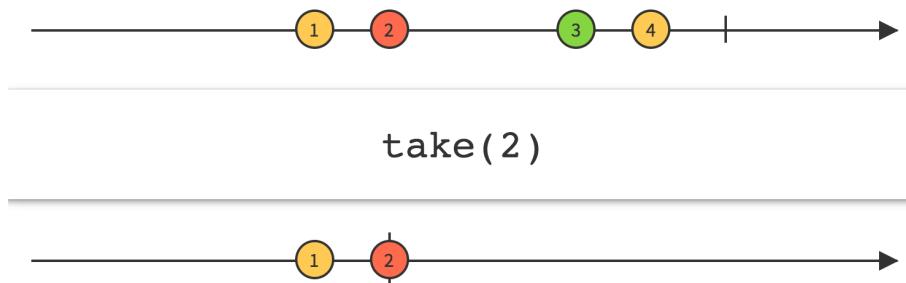
注意：这个机制是这样实现的：延迟原始 Observable' 发射的任何数据项，直到这次发射之后过了给定的时长。

`skipLast`的这个变体默认在`computation`调度器上执行，但是你可以使用第三个参数指定其它的调度器。

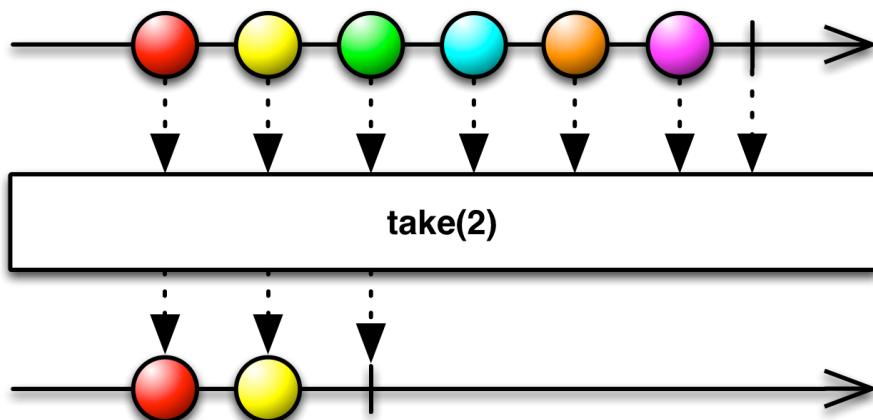
- Javadoc: `skipLast(long, TimeUnit)`
- Javadoc: `skipLast(long, TimeUnit, Scheduler)`

Take

只发射前面的N项数据



使用 `Take` 操作符让你可以修改 Observable 的行为，只返回前面的 N 项数据，然后发射完成通知，忽略剩余的数据。



RxJava将这个操作符实现为 `take` 函数。

如果你对一个Observable使用 `take(n)`（或它的同义词 `limit(n)`）操作符，而那个Observable发射的数据少于N项，那么 `take` 操作生成的Observable不会抛异常或发射 `onError` 通知，在完成前它只会发射相同的少量数据。

示例代码

```
observable.just(1, 2, 3, 4, 5, 6, 7, 8)
    .take(4)
    .subscribe(new Subscriber<Integer>() {
        @Override
        public void onNext(Integer item) {
            System.out.println("Next: " + item);
        }
    })
```

```

        public void onError(Throwable error) {
            System.err.println("Error: " +
error.getMessage());
        }

        @Override
        public void onCompleted() {
            System.out.println("Sequence complete.");
        }
    });

```

输出

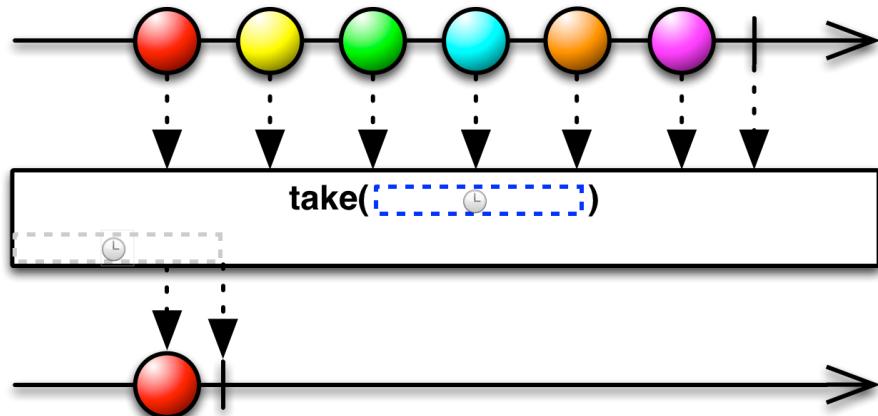
```

Next: 1
Next: 2
Next: 3
Next: 4
Sequence complete.

```

`take(int)` 默认不任何特定的调度器上执行。

- Javadoc: [take\(int\)](#)



`take` 的这个变体接受一个时长而不是数量参数。它会丢发射 Observable 开始的那段时间发射的数据，时长和时间单位通过参数指定。

`take` 的这个变体默认在 `computation` 调度器上执行，但是你可以使用第三个参数指定其它的调度器。

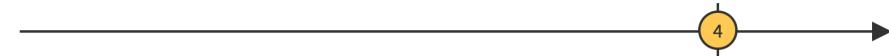
- Javadoc: [take\(long, TimeUnit\)](#)
- Javadoc: [take\(long, TimeUnit, Scheduler\)](#)

TakeLast

发射 Observable 发射的最后 N 项数据

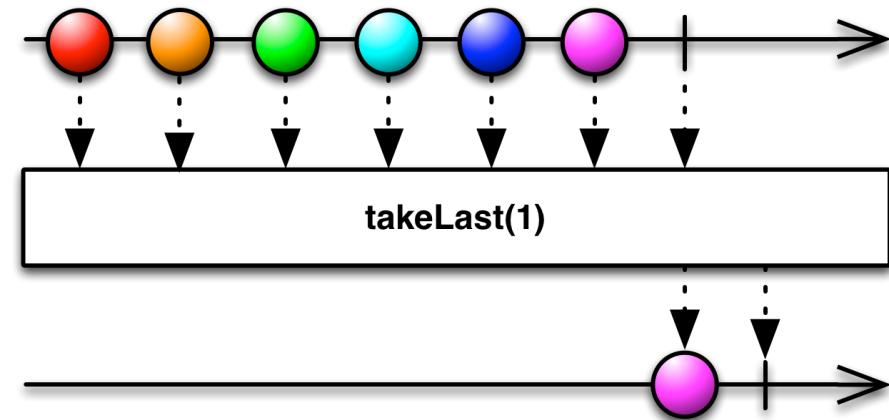


`takeLast(1)`



使用 `TakeLast` 操作符修改原始 Observable，你可以只发射 Observable' 发射的后 N 项数据，忽略前面的数据。

`taskLast.n`

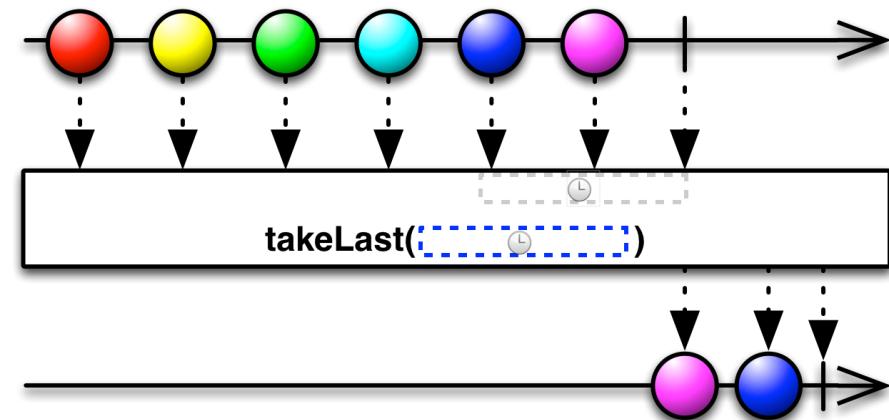


使用 `takeLast` 操作符，你可以只发射原始 Observable' 发射的后 N 项数据，忽略之前的数据。注意：这会延迟原始 Observable' 发射的任何数据项，直到它全部完成。

`takeLast` 的这个变体默认不在任何特定的调度器上执行。

- Javadoc: [takeLast\(int\)](#)

`takeLast.t`

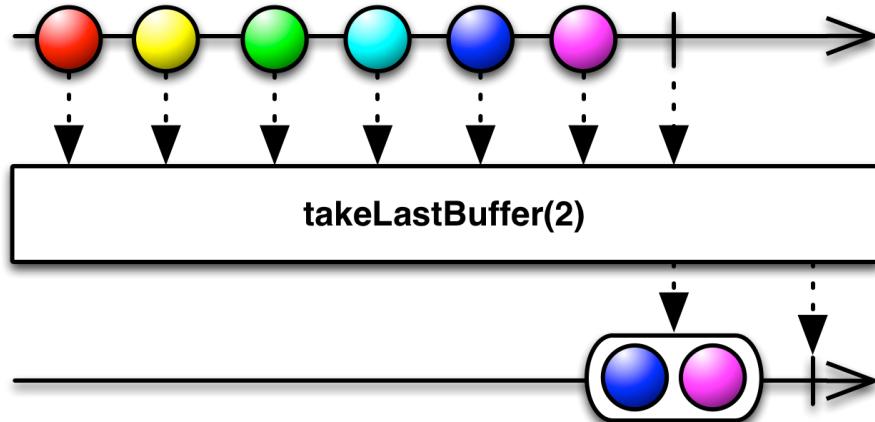


还有一个 `takeLast` 变体接受一个时长而不是数量参数。它会发射在原始 Observable' 的生命周期内最后一段时间内发射的数据。时长和时间单位通过参数指定。

注意：这会延迟原始Observable发射的任何数据项，直到它全部完成。

`takeLast`的这个变体默认在`computation`调度器上执行，但是你可以使用第三个参数指定其它的调度器。

takeLastBuffer



还有一个操作符叫`takeLastBuffer`，它和`takeLast`类似，，唯一的不同是它把所有的数据项收集到一个`List`再发射，而不是依次发射一个。

- Javadoc: `takeLastBuffer(int)`
- Javadoc: `takeLastBuffer(long,TimeUnit)`
- Javadoc: `takeLastBuffer(long,TimeUnit,Scheduler)`
- Javadoc: `takeLastBuffer(int,long,TimeUnit)`
- Javadoc: `takeLastBuffer(int,long,TimeUnit,Scheduler)`

结合操作

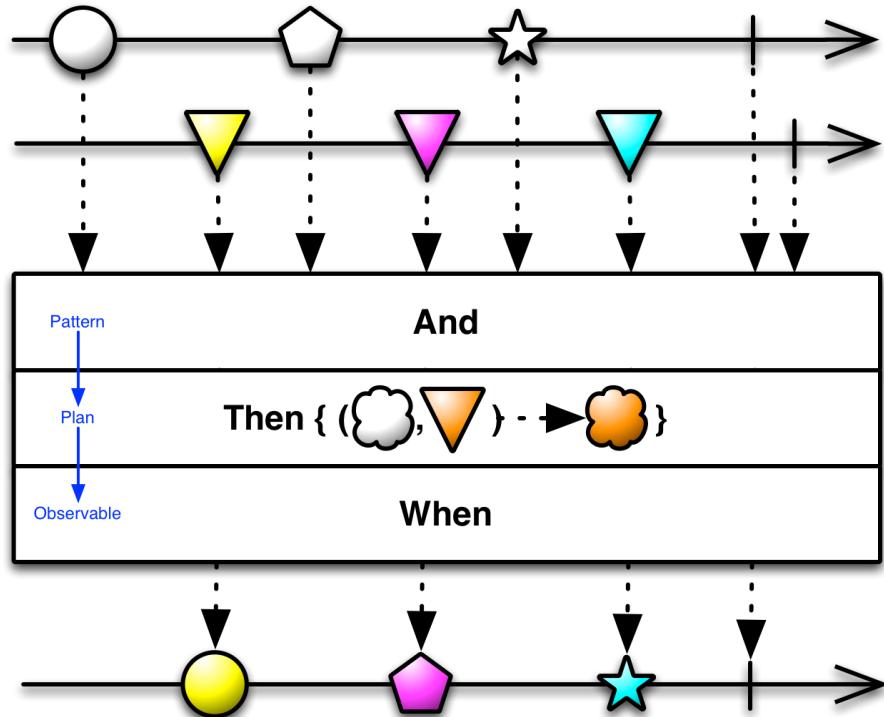
这个页面展示的操作符可用于组合多个Observables。

- **startWith()** – 在数据序列的开头增加一项数据
- **merge()** – 将多个Observable合并为一个
- **mergeDelayError()** – 合并多个Observables，让没有错误的 Observable都完成后再发射错误通知
- **zip()** – 使用一个函数组合多个Observable发射的数据集合，然后再发射这个结果
- **and(), then(), and when()** – (`rxjava-joins`) 通过模式和计划组合多个Observables发射的数据集合
- **combineLatest()** – 当两个Observables中的任何一个发射了一个数据时，通过一个指定的函数组合每个Observable发射的最新数据（一共两个数据），然后发射这个函数的结果
- **join() and groupJoin()** – 无论何时，如果一个Observable发射了一个数据项，只要在另一个Observable发射的数据项定义的时间窗口内，就将两个Observable发射的数据合并发射
- **switchOnNext()** – 将一个发射Observables的Observable转换成另一个Observable，后者发射这些Observables最近发射的数据

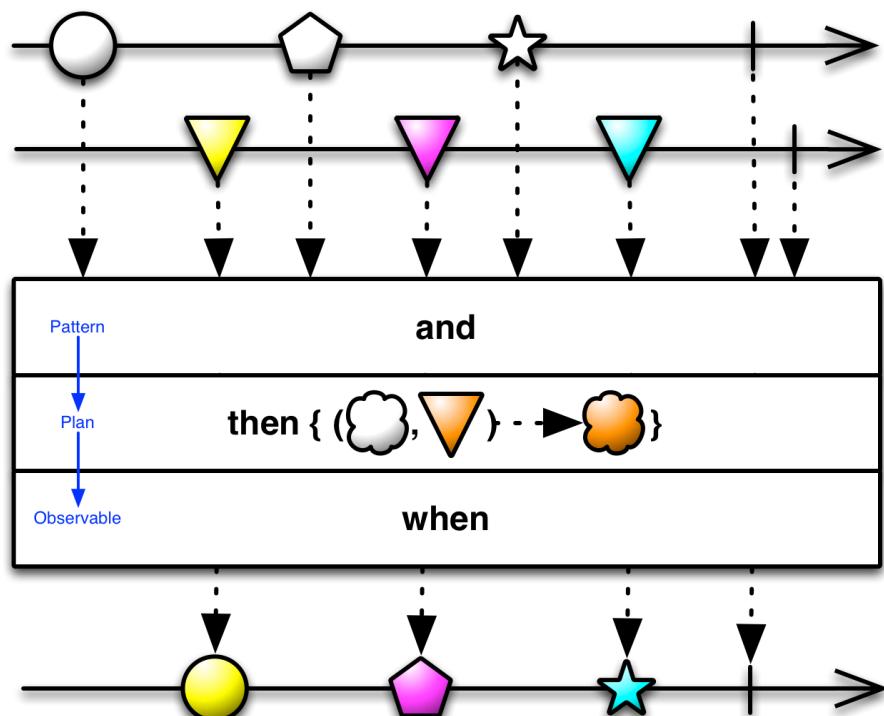
(*rxjava-joins*) — 表示这个操作符当前是可选的*rxjava-joins*包的一部分，还没有包含在标准的RxJava操作符集合里

And/Then/When

使用Pattern和Plan作为中介，将两个或多个Observable发射的数据集合并到一起



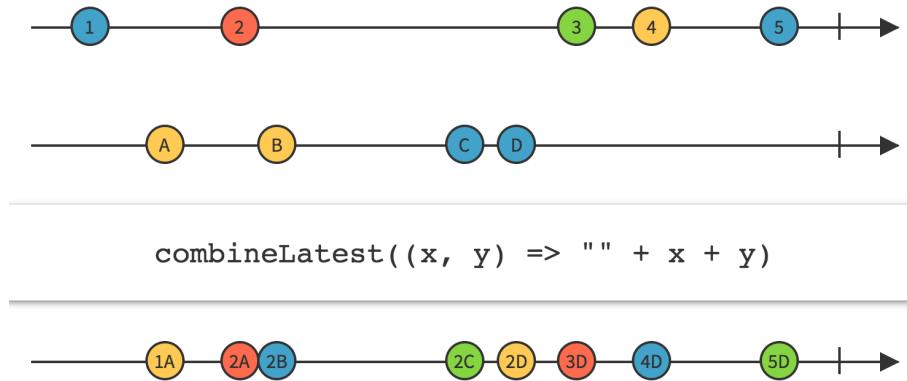
And/Then/When操作符组合的行为类似于[zip](#)，但是它们使用一个中间数据结构。接受两个或多个Observable，一次一个将它们的发射物合并到Pattern对象，然后操作那个Pattern对象，变换为一个Plan。随后将这些Plan变换为Observable的发射物。



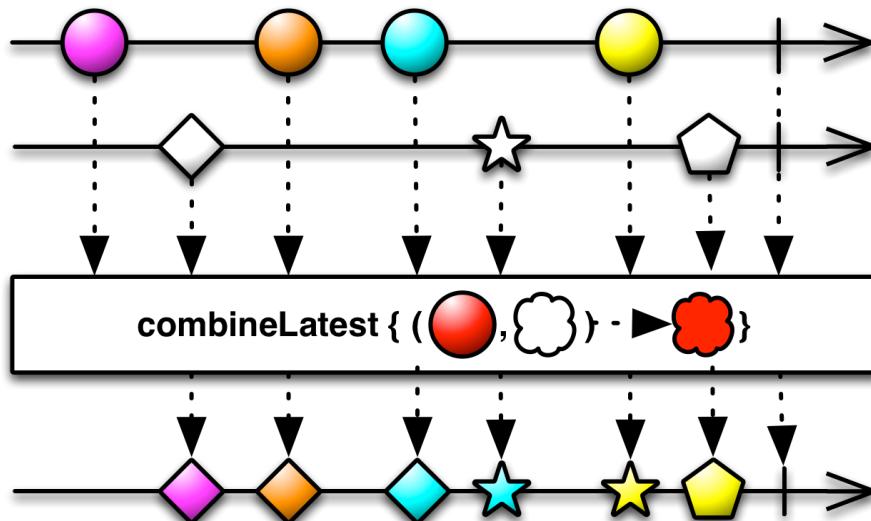
它们属于 `rxjava-joins` 模块，不是核心RxJava包的一部分

CombineLatest

当两个Observables中的任何一个发射了数据时，使用一个函数结合每个Observable发射的最近数据项，并且基于这个函数的结果发射数据。



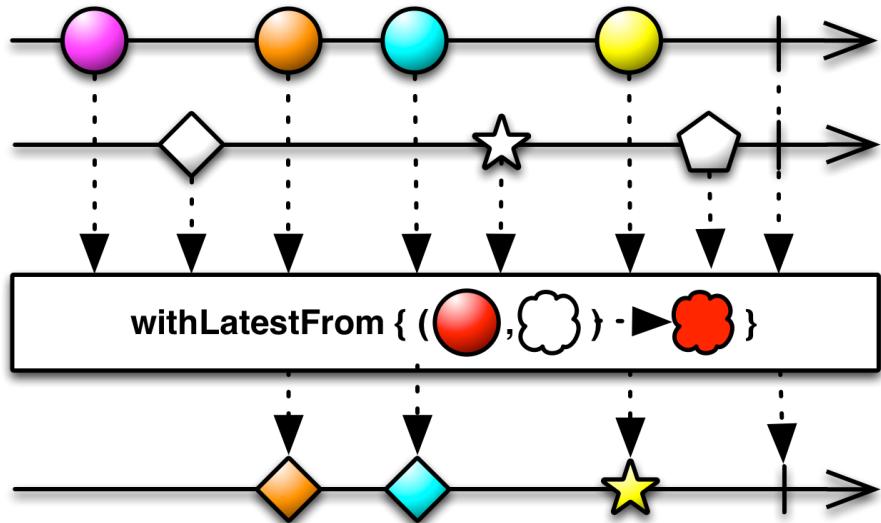
`CombineLatest` 操作符行为类似于 `zip`，但是只有当原始的Observable中的每一个都发射了一条数据时 `zip` 才发射数据。`CombineLatest` 则在原始的Observable中任意一个发射了数据时发射一条数据。当原始Observables的任何一个发射了一条数据时，`CombineLatest` 使用一个函数结合它们最近发射的数据，然后发射这个函数的返回值。



RxJava将这个操作符实现为 `combineLatest`，它接受二到九个Observable作为参数，或者单个Observables列表作为参数。它默认不在任何特定的调度器上执行。

- Javadoc: `combineLatest(List,FuncN)`
- Javadoc: `combineLatest(Observable,Observable,Func2)`

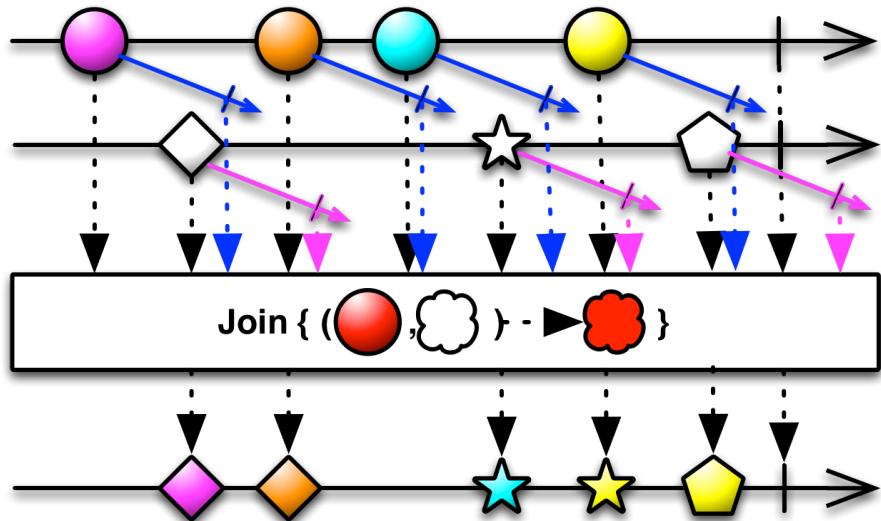
withLatestFrom



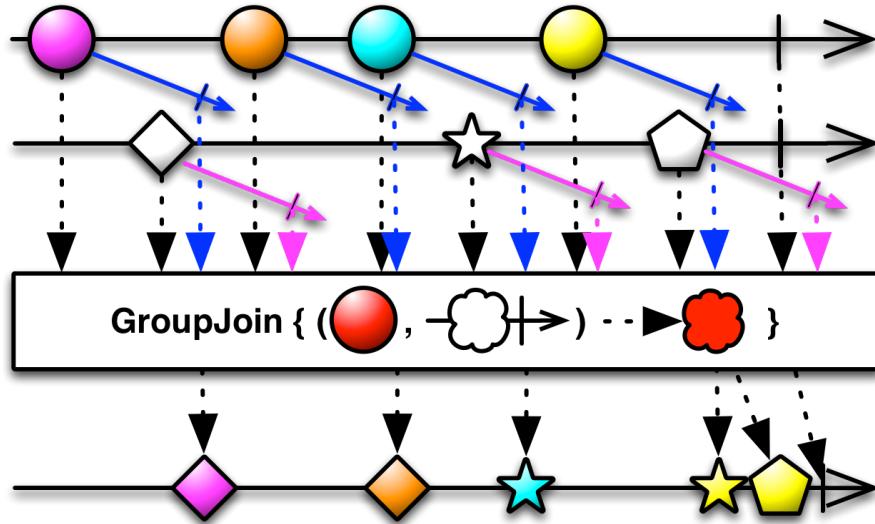
`withLatestFrom` 操作符还在开发中，不是1.0版本的一部分。类似于 `combineLatest`，但是只在单个原始Observable发射了一条数据时才发射数据。

Join

任何时候，只要在另一个Observable发射的数据定义的时间窗口内，这个 Observable发射了一条数据，就结合两个Observable发射的数据。

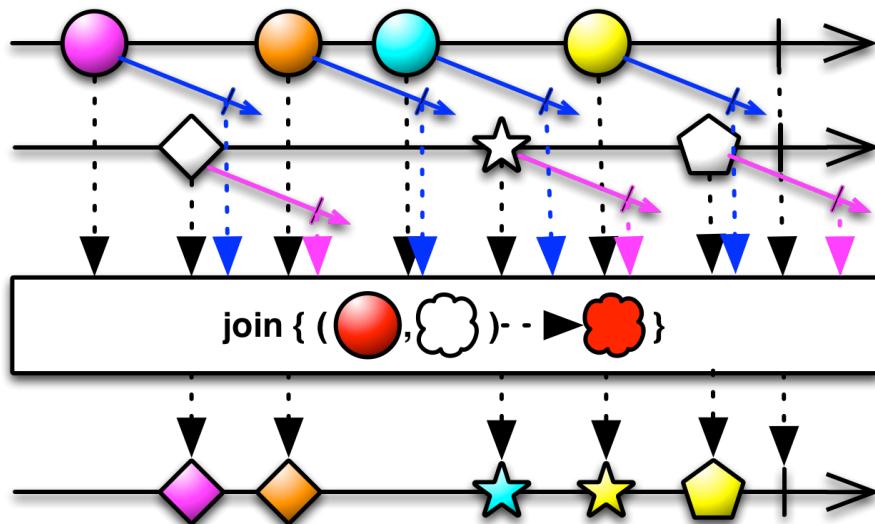


`Join` 操作符结合两个Observable发射的数据，基于时间窗口（你定义的针对每条数据特定的原则）选择待集合的数据项。你将这些时间窗口实现为一些 Observables，它们的生命周期从任何一条Observable发射的每一条数据开始。当这个定义时间窗口的Observable发射了一条数据或者完成时，与这条数据关联的窗口也会关闭。只要这条数据的窗口是打开的，它将继续结合其它Observable 发射的任何数据项。你定义一个用于结合数据的函数。



很多ReactiveX实现还有一个类似的**GroupJoin**操作符。

Most ReactiveX implementations that have a Join operator also have a GroupJoin operator that is similar, except that the function you define to combine items emitted by the two Observables pairs individual items emitted by the source Observable not with an item from the second Observable, but with an Observable that emits items from the second Observable that fall in the same window.

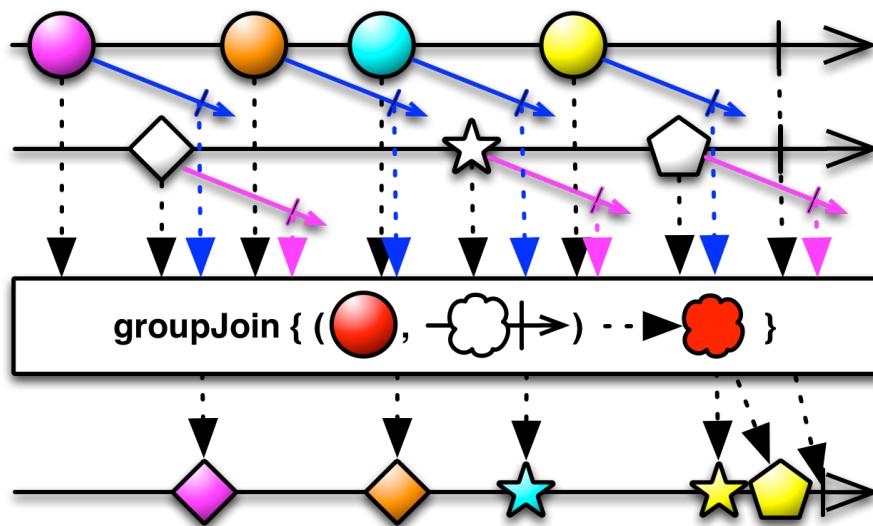


The `join` operator takes four parameters:

1. the second Observable to combine with the source Observable
2. a function that accepts an item from the source Observable and returns an Observable whose lifespan governs the duration during which that item will combine with items from the second Observable
3. a function that accepts an item from the second Observable and returns an Observable whose lifespan governs the duration during which that item will combine with items from the first Observable
4. a function that accepts an item from the first Observable and an item from the second Observable and returns an item to be emitted by the Observable returned from join

`join`默认不在任何特定的调度器上执行。

- Javadoc: Join(Observable,Func1,Func1,Func2)

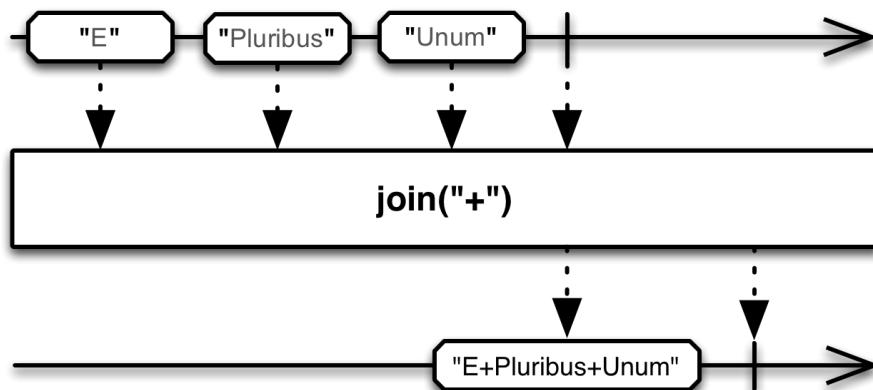


The `groupJoin` operator takes four parameters:

1. the second Observable to combine with the source Observable
2. a function that accepts an item from the source Observable and returns an Observable whose lifespan governs the duration during which that item will combine with items from the second Observable
3. a function that accepts an item from the second Observable and returns an Observable whose lifespan governs the duration during which that item will combine with items from the first Observable
4. a function that accepts an item from the first Observable and an Observable that emits items from the second Observable and returns an item to be emitted by the Observable returned from `groupJoin`

`groupJoin`默认不在任何特定的调度器上执行。

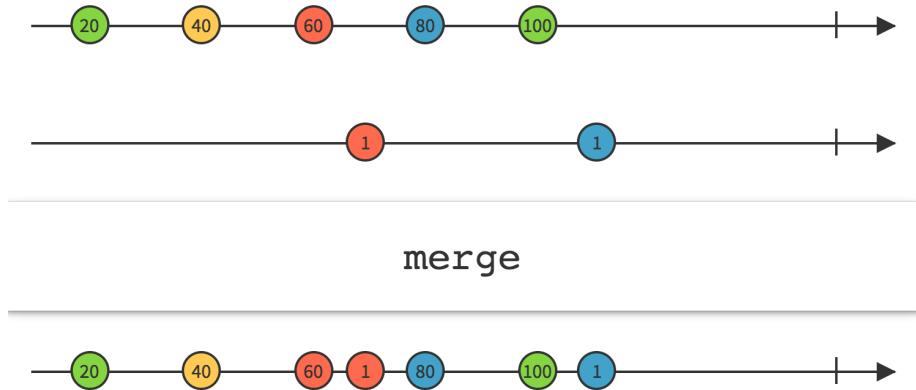
- Javadoc: groupJoin(Observable,Func1,Func1,Func2)



可选的 `StringObservable` 类中也有一个 `join` 操作符。它将一个发射字符串序列的 Observable 转换为一个发射单个字符串的 Observable，`join` 操作符使用指定的定界符将全部单独的字符串连接起来。

Merge

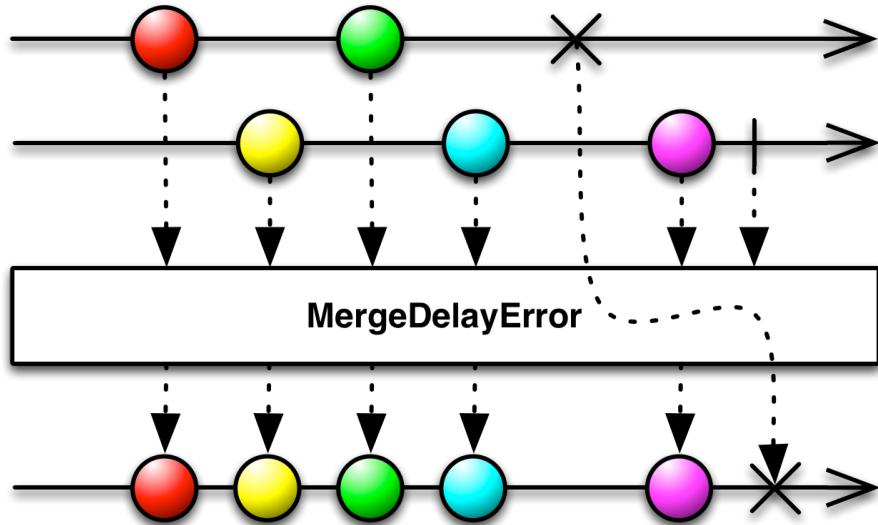
合并多个Observables的发射物



使用 `Merge` 操作符你可以将多个Observables的输出合并，就好像它们是一个单个的Observable一样。

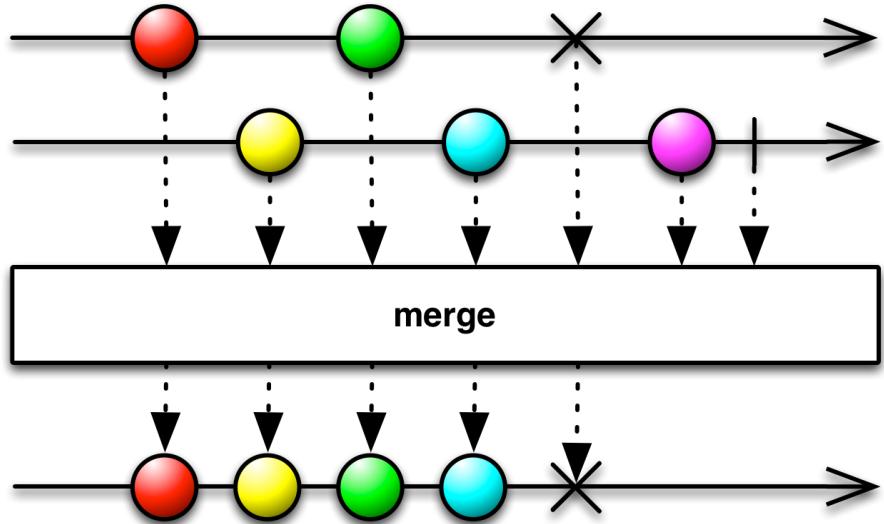
`Merge` 可能会让合并的Observables发射的数据交错（有一个类似的操作符 `Concat` 不会让数据交错，它会按顺序一个接着一个发射多个Observables的发射物）。

正如图例上展示的，任何一个原始Observable的 `onError` 通知会被立即传递给观察者，而且会终止合并后的Observable。



在很多ReactiveX实现中还有一个叫 `MergeDelayError` 的操作符，它的行为有一点不同，它会保留 `onError` 通知直到合并后的Observable所有的数据发射完成，在那时它才会把 `onError` 传递给观察者。

RxJava将它实现为 `merge`, `mergewith` 和 `mergeDelayError`。



示例代码

```

observable<Integer> odds = Observable.just(1, 3,
5).subscribeOn(somescheduler);
Observable<Integer> evens = Observable.just(2, 4, 6);

Observable.merge(odds, evens)
    .subscribe(new Subscriber<Integer>() {
        @Override
        public void onNext(Integer item) {
            System.out.println("Next: " + item);
        }

        @Override
        public void onError(Throwable error) {
            System.err.println("Error: " +
error.getMessage());
        }

        @Override
        public void onCompleted() {
            System.out.println("Sequence complete.");
        }
    });

```

输出

```

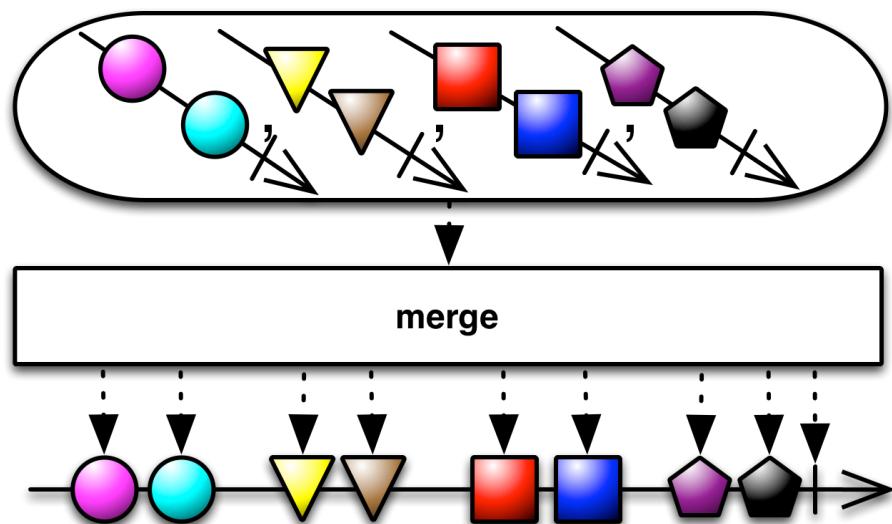
Next: 1
Next: 3
Next: 5
Next: 2
Next: 4
Next: 6
Sequence complete.

```

- Javadoc: `merge(Iterable)`

- Javadoc: `merge(Iterable,int)`
- Javadoc: `[merge(Observable)]`
- Javadoc: `merge(Observable/Observable)` (接受二到九个Observable)

除了传递多个Observable给`merge`，你还可以传递一个Observable列表`List`，数组，甚至是一个发射Observable序列的Observable，`merge`将合并它们的输出作为单个Observable的输出：

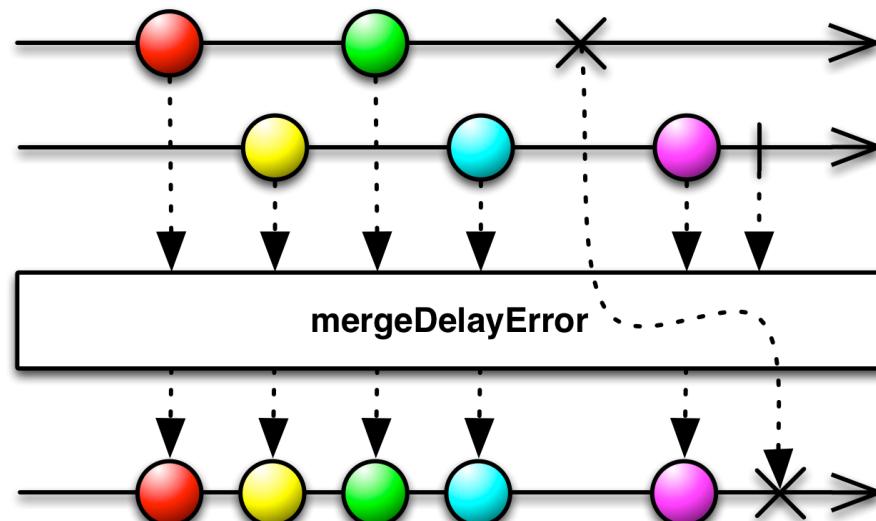


如果你传递一个发射Observables序列的Observable，你可以指定`merge`应该同时订阅的Observable的最大数量。一旦达到订阅数的限制，它将不再订阅原始Observable发射的任何其它Observable，直到某个已经订阅的Observable发射了`onCompleted`通知。

- Javadoc: `merge(Observable)`
- Javadoc: `merge(Observable,int)`

`merge`是静态方法，`mergeWith`是对象方法，举个例子，
`Observable.merge(odds, evens)`等价于`odds.mergeWith(evens)`。

如果传递给`merge`的任何一个的Observable发射了`onError`通知终止了，`merge`操作符生成的Observable也会立即以`onError`通知终止。如果你想让它继续发射数据，在最后才报告错误，可以使用`mergeDelayError`。



`mergeDelayError` behaves much like `merge`. The exception is when one of the Observables being merged terminates with an `onError` notification. If this happens with `merge`, the merged Observable will immediately issue an `onError` notification and terminate. `mergeDelayError`, on the other hand, will hold off on reporting the error until it has given any other non-error-producing Observables that it is merging a chance to finish emitting their items, and it will emit those itself, and will only terminate with an `onError` notification when all of the other merged Observables have finished.

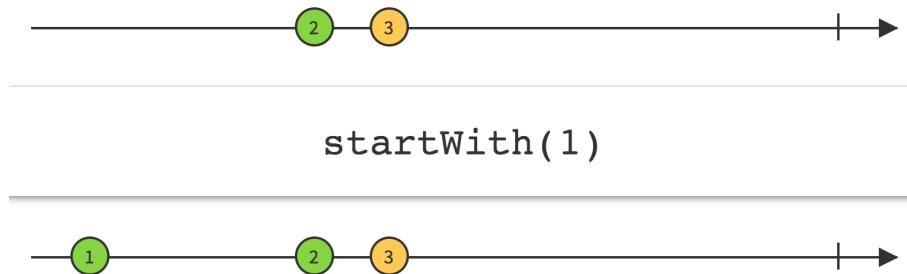
Because it is possible that more than one of the merged Observables encountered an error, `mergeDelayError` may pass information about multiple errors in the `onError` notification (it will never invoke the observer's `onError` method more than once). For this reason, if you want to know the nature of these errors, you should write your observers' `onError` methods so that they accept a parameter of the class `CompositeException`.

`mergeDelayError` has fewer variants. You cannot pass it an Iterable or Array of Observables, but you can pass it an Observable that emits Observables or between one and nine individual Observables as parameters. There is not an instance method version of `mergeDelayError` as there is for `merge`.

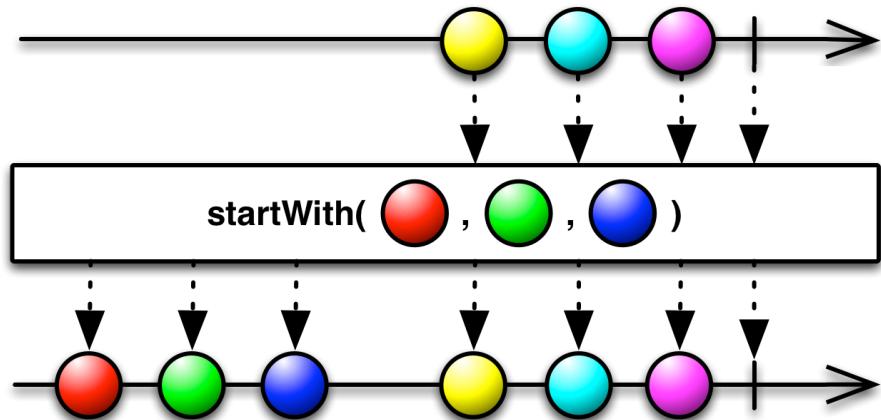
- Javadoc: `mergeDelayError(Observable)`
- Javadoc: `mergeDelayError(Observable, Observable)`

StartWith

在数据序列的开头插入一条指定的项

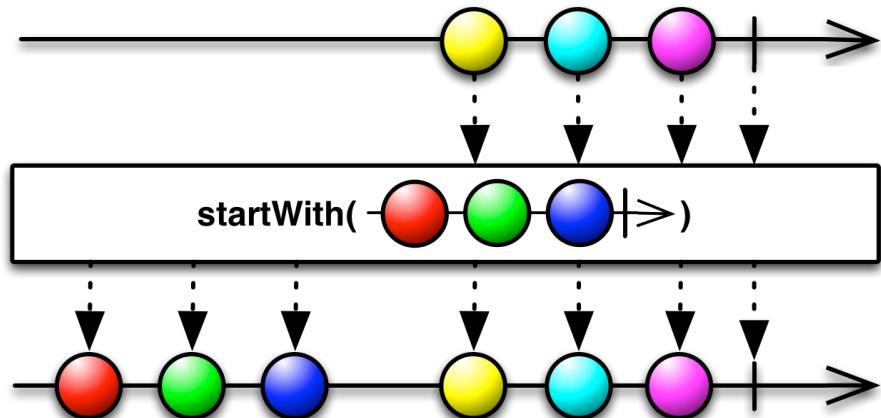


如果你想要一个Observable在发射数据之前先发射一个指定的数据序列，可以使用`Startwith`操作符。（如果你想一个Observable发射的数据末尾追加一个数据序列可以使用`Concat`操作符。）



可接受一个Iterable或者多个Observable作为函数的参数。

- Javadoc: `startWith(Iterable)`
- Javadoc: `startWith(T)` (最多接受九个参数)

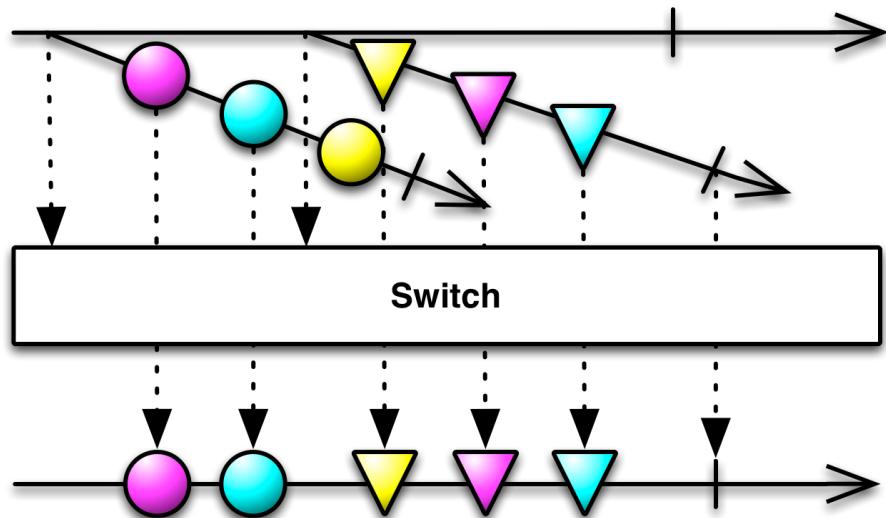


你也可以传递一个Observable给 `startWith`，它会将那个Observable的发射物插在原始Observable发射的数据序列之前，然后把这个当做自己的发射物集合。这可以看作是 `Concat` 的反转。

- Javadoc: `startWith(Observable)`

Switch

将一个发射多个Observables的Observable转换成另一个单独的Observable，后者发射那些Observables最近发射的数据项



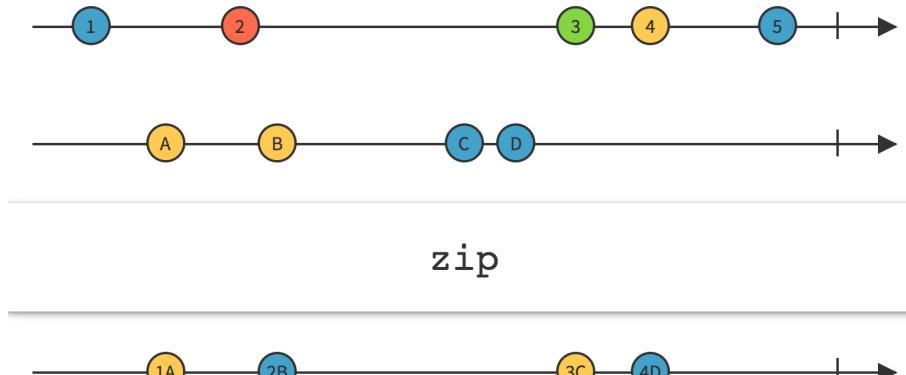
`switch` 订阅一个发射多个Observables的Observable。它每次观察那些Observables中的一个，`switch`返回的这个Observable取消订阅前一个发射数据的Observable，开始发射最近的Observable发射的数据。注意：当原始Observable发射了一个新的Observable时（不是这个新的Observable发射了一条数据时），它将取消订阅之前的那个Observable。这意味着，在后来那个Observable产生之后到它开始发射数据之前的这段时间里，前一个Observable发射的数据将被丢弃（就像图例上的那个黄色圆圈一样）。

Java将这个操作符实现为`switchOnNext`。它默认不在任何特定的调度器上执行。

- Javadoc: [switchOnNext\(Observable\)](#)

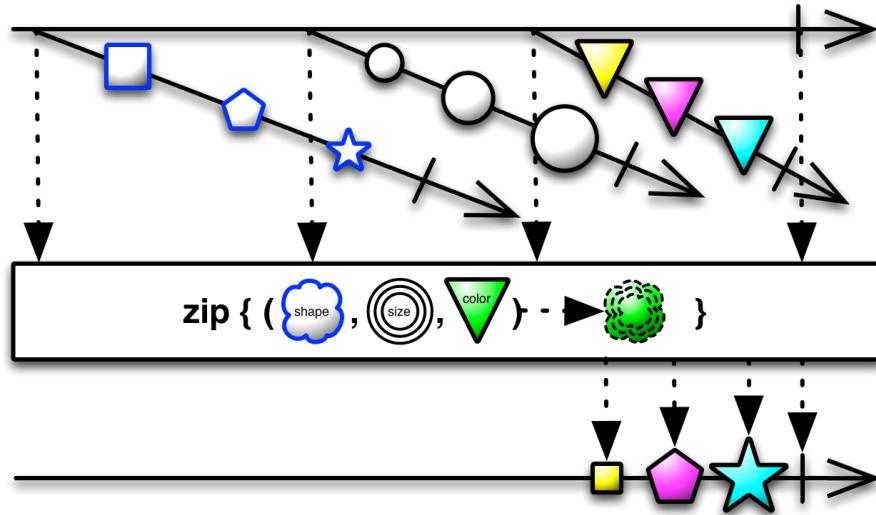
Zip

通过一个函数将多个Observables的发射物结合到一起，基于这个函数的结果为每个结合体发射单个数据项。



`zip`操作符返回一个Obversable，它使用这个函数按顺序结合两个或多个Observables发射的数据项，然后它发射这个函数返回的结果。它按照严格的顺序应用这个函数。它只发射与发射数据项最少的那个Observable一样多的数据。

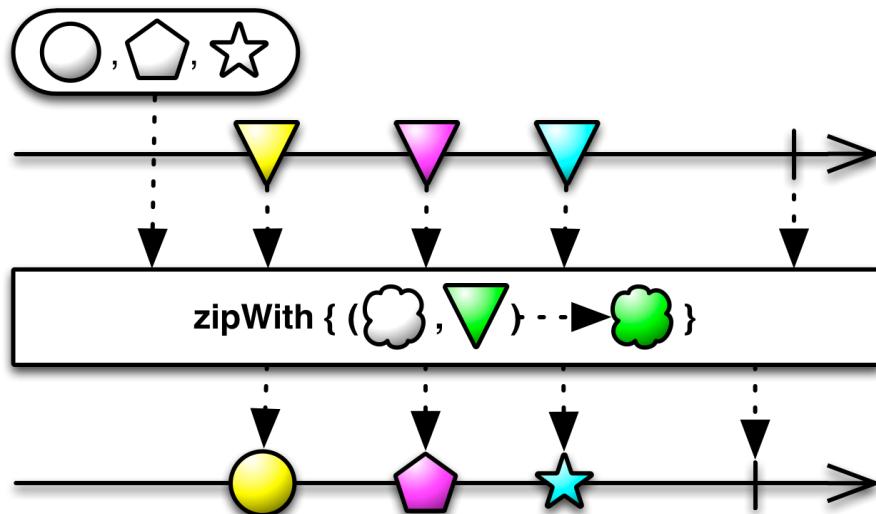
RxJava将这个操作符实现为`zip`和`zipWith`。



`zip`的最后一个参数接受每个Observable发射的一项数据，返回被压缩后的数据，它可以接受一到九个参数：一个Observable序列，或者一些发射Observable的Observables。

- Javadoc: `zip(Iterable,FuncN)`
- Javadoc: `zip(Observable,FuncN)`
- Javadoc: `zip(Observable/Observable,Func2)` (最多可以有九个Observables参数)

zipWith



`zipWith`操作符总是接受两个参数，第一个参数是一个Observable或者一个Iterable。

- Javadoc: `zipWith(Observable,Func2)`
- Javadoc: `zipWith(Iterable,Func2)`

`zip`和`zipwith`默认不在任何特定的操作符上执行。

错误处理

很多操作符可用于对Observable发射的onError通知做出响应或者从错误中恢复，例如，你可以：

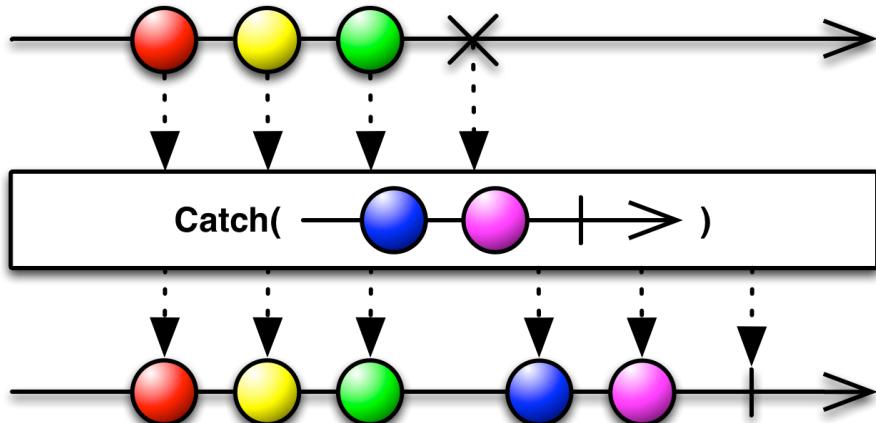
1. 吞掉这个错误，切换到一个备用的Observable继续发射数据
2. 吞掉这个错误然后发射默认值
3. 吞掉这个错误并立即尝试重启这个Observable
4. 吞掉这个错误，在一些回退间隔后重启这个Observable

这是操作符列表：

- **onErrorResumeNext()** – 指示Observable在遇到错误时发射一个数据序列
- **onErrorReturn()** – 指示Observable在遇到错误时发射一个特定的数据
- **onExceptionResumeNext()** – instructs an Observable to continue emitting items after it encounters an exception (but not another variety of throwable) 指示Observable遇到错误时继续发射数据
- **retry()** – 指示Observable遇到错误时重试
- **retryWhen()** – 指示Observable遇到错误时，将错误传递给另一个Observable来决定是否要重新给订阅这个Observable

Catch

从onError通知中恢复发射数据



Catch操作符拦截原始Observable的onError通知，将它替换为其它的数据项或数据序列，让产生的Observable能够正常终止或者根本不终止。

在某些ReactiveX的实现中，有一个叫**onErrorResumeNext**的操作符，它的行为与**Catch**相似。

RxJava将**Catch**实现为三个不同的操作符：

onErrorReturn

让Observable遇到错误时发射一个特殊的项并且正常终止。

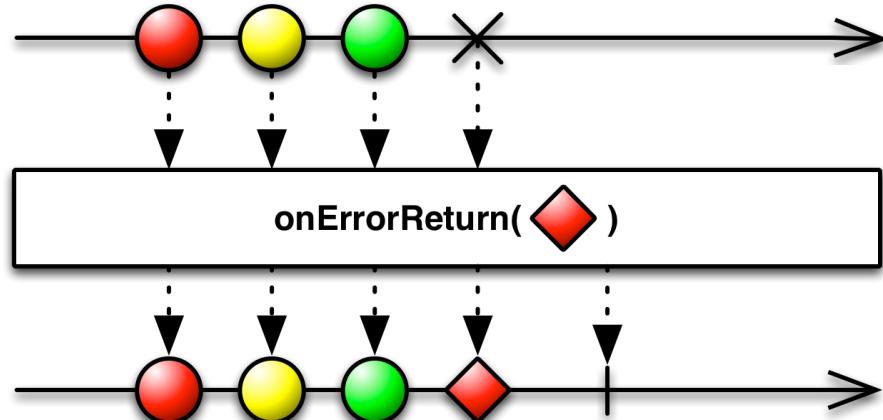
onErrorResumeNext

让Observable在遇到错误时开始发射第二个Observable的数据序列。

onExceptionResumeNext

让Observable在遇到错误时继续发射后面的数据项。

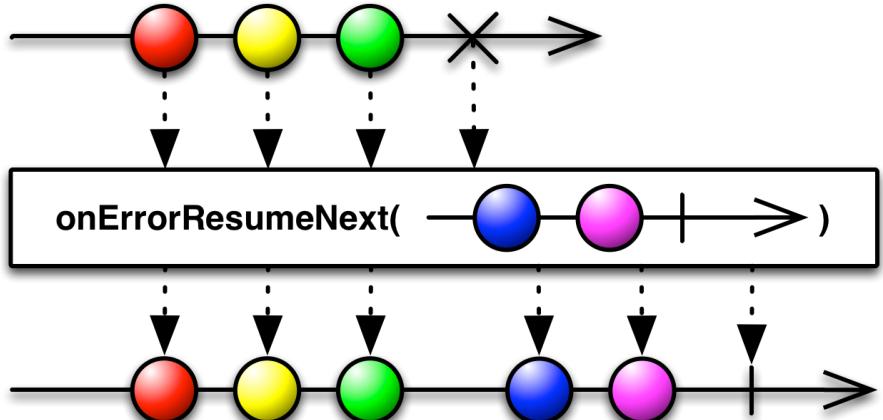
onErrorReturn



`onErrorReturn`方法返回一个镜像原有Observable行为的新Observable，后者会忽略前者的`onError`调用，不会将错误传递给观察者，作为替代，它会发发射一个特殊的项并调用观察者的`onCompleted`方法。

- Javadoc: [onErrorReturn\(Func1\)](#)

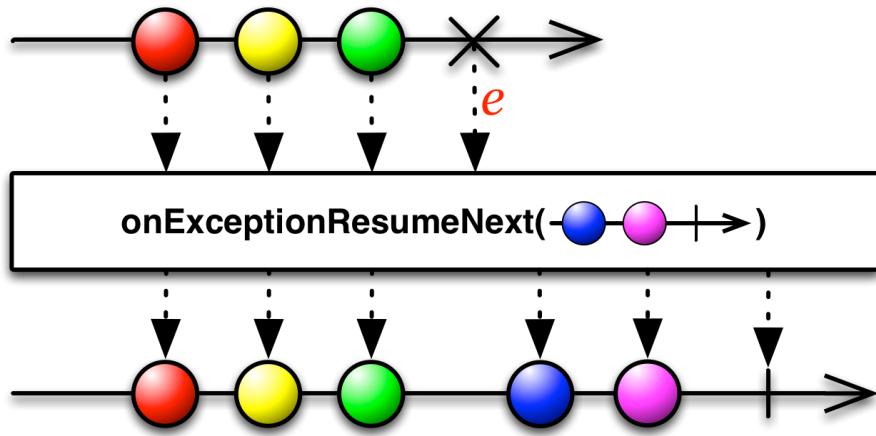
onErrorResumeNext



`onErrorResumeNext`方法返回一个镜像原有Observable行为的新Observable，后者会忽略前者的`onError`调用，不会将错误传递给观察者，作为替代，它会开始镜像另一个，备用的Observable。

- Javadoc: [onErrorResumeNext\(Func1\)](#)
- Javadoc: [onErrorResumeNext\(Observable\)](#)

onExceptionResumeNext

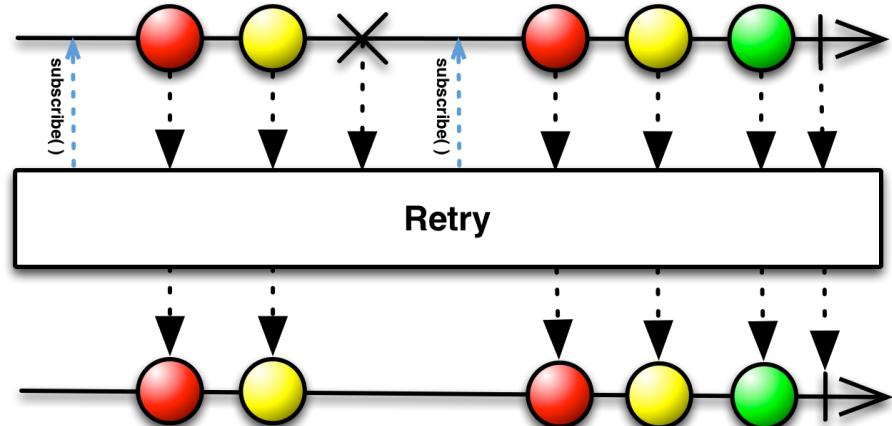


和 `onErrorResumeNext` 类似，`onExceptionResumeNext` 方法返回一个镜像原有 Observable 行为的新 Observable，也使用一个备用的 Observable，不同的是，如果 `onError` 收到的 `Throwable` 不是一个 `Exception`，它会将错误传递给观察者的 `onError` 方法，不会使用备用的 Observable。

- Javadoc: `onExceptionResumeNext(Observable)`

Retry

如果原始 Observable 遇到错误，重新订阅它期望它能正常终止



`Retry` 操作符不会将原始 Observable 的 `onError` 通知传递给观察者，它会订阅这个 Observable，再给它一次机会无错误地完成它的数据序列。`Retry` 总是传递 `onNext` 通知给观察者，由于重新订阅，可能会造成数据项重复，如上图所示。

RxJava 中的实现为 `retry` 和 `retrywhen`。

无论收到多少次 `onError` 通知，无参数版本的 `retry` 都会继续订阅并发射原始 Observable。

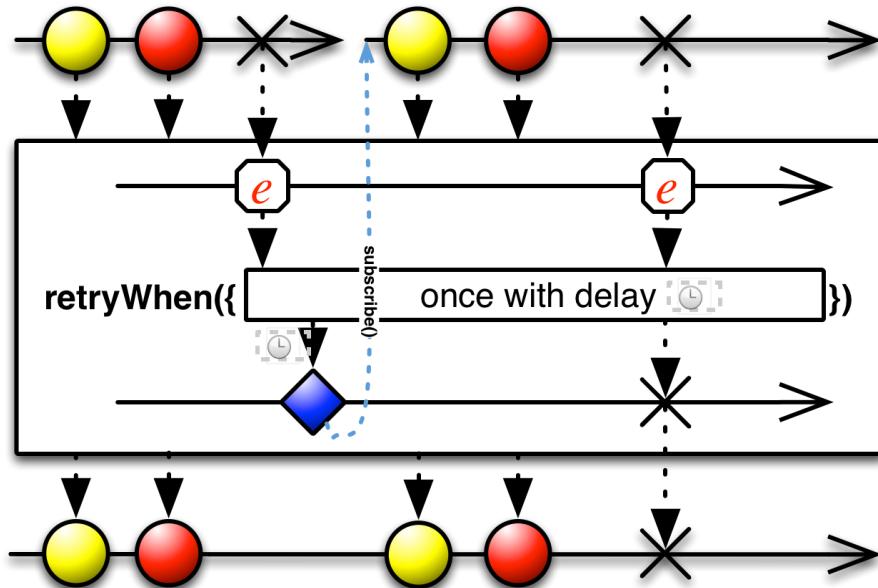
接受单个 `count` 参数的 `retry` 会最多重新订阅指定的次数，如果次数超了，它不会尝试再次订阅，它会把最新的一个 `onError` 通知传递给它的观察者。

还有一个版本的 `retry` 接受一个谓词函数作为参数，这个函数的两个参数是：重试次数和导致发射 `onError` 通知的 `Throwable`。这个函数返回一个布尔值，如果返回 `true`，`retry` 应该再次订阅和镜像原始的 Observable，如果返回 `false`，`retry` 会将最新的一个 `onError` 通知传递给它的观察者。

`retry`操作符默认在`trampoline`调度器上执行。

- Javadoc: `retry()`
- Javadoc: `retry(long)`
- Javadoc: `retry(Func2)`

retryWhen



`retrywhen`和`retry`类似，区别是，`retrywhen`将`onError`中的`Throwable`传递给一个函数，这个函数产生另一个Observable，`retrywhen`观察它的结果再决定是不是要重新订阅原始的Observable。如果这个Observable发射了一项数据，它就重新订阅，如果这个Observable发射的是`onError`通知，它就将这个通知传递给观察者然后终止。

`retrywhen`默认在`trampoline`调度器上执行，你可以通过参数指定其它的调度器。

示例代码

```
observable.create((Subscriber<? super String> s) -> {
    System.out.println("subscribing");
    s.onError(new RuntimeException("always fails"));
}).retryWhen(attempts -> {
    return attempts.zipwith(Observable.range(1, 3), (n,
    i) -> i).flatMap(i -> {
        System.out.println("delay retry by " + i + " second(s)");
        return observable.timer(i, TimeUnit.SECONDS);
    });
}).toBlocking().forEach(System.out::println);
```

输出

```
subscribing
delay retry by 1 second(s)
subscribing
delay retry by 2 second(s)
subscribing
delay retry by 3 second(s)
subscribing
```

- Javadoc: `retryWhen(Func1)`
- Javadoc: `retryWhen(Func1,Scheduler)`

辅助操作

这个页面列出了很多用于Observable的辅助操作符

- **materialize()** – 将Observable转换成一个通知列表convert an Observable into a list of Notifications
- **dematerialize()** – 将上面的结果逆转变回一个Observable
- **timestamp()** – 给Observable发射的每个数据项添加一个时间戳
- **serialize()** – 强制Observable按次序发射数据并且要求功能是完好的
- **cache()** – 记住Observable发射的数据序列并发射相同的数据序列给后续的订阅者
- **observeOn()** – 指定观察者观察Observable的调度器
- **subscribeOn()** – 指定Observable执行任务的调度器
- **doOnEach()** – 注册一个动作，对Observable发射的每个数据项使用
- **doOnCompleted()** – 注册一个动作，对正常完成的Observable使用
- **doOnError()** – 注册一个动作，对发生错误的Observable使用
- **doOnTerminate()** – 注册一个动作，对完成的Observable使用，无论是否发生错误
- **doOnSubscribe()** – 注册一个动作，在观察者订阅时使用
- **doOnUnsubscribe()** – 注册一个动作，在观察者取消订阅时使用
- **finallyDo()** – 注册一个动作，在Observable完成时使用
- **delay()** – 延时发射Observable的结果
- **delaySubscription()** – 延时处理订阅请求
- **timeInterval()** – 定期发射数据
- **using()** – 创建一个只在Observable生命周期存在的资源
- **single()** – 强制返回单个数据，否则抛出异常
- **singleOrDefault()** – 如果Observable完成时返回了单个数据，就返回它，否则返回默认数据
- **toFuture(), toIterable(), toList()** – 将Observable转换为其它对象或数据结构

Delay

延迟一段指定的时间再发射来自Observable的发射物

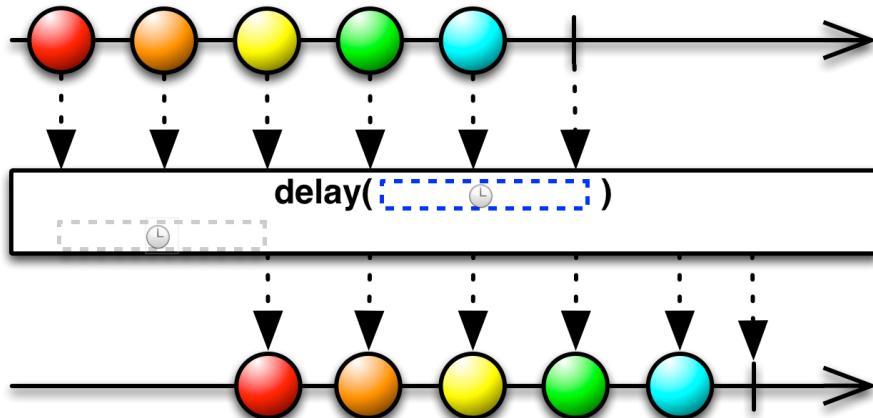


delay



`delay`操作符让原始Observable在发射每项数据之前都暂停一段指定的时间段。效果是Observable发射的数据项在时间上向前整体平移了一个增量。

RxJava的实现是 `delay` 和 `delaySubscription`。

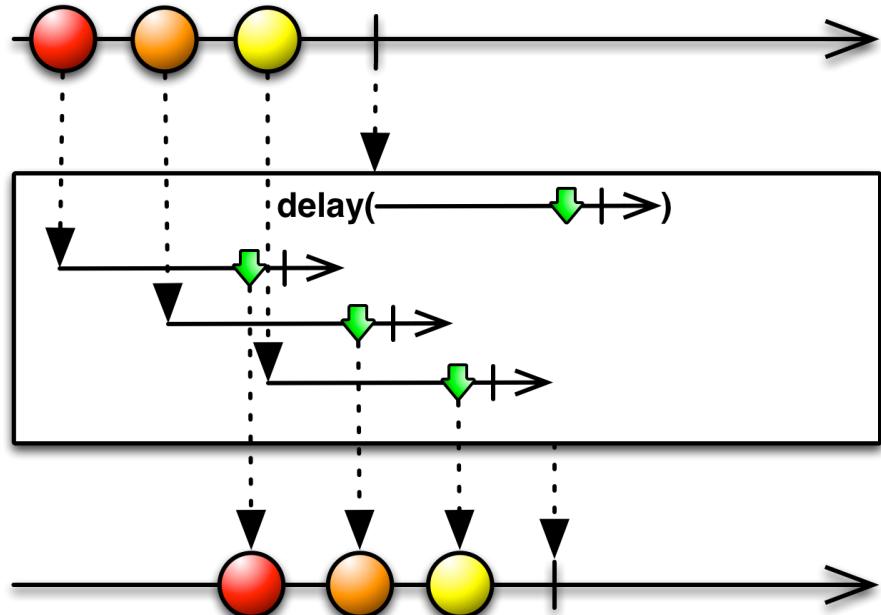


第一种 `delay` 接受一个定义时长的参数（包括数量和单位）。每当原始 Observable 发射一项数据，`delay` 就启动一个定时器，当定时器过了给定的时间段时，`delay` 返回的 Observable 发射相同的数据项。

注意：`delay` 不会平移 `onError` 通知，它会立即将这个通知传递给订阅者，同时丢弃任何待发射的 `onNext` 通知。然而它会平移一个 `onCompleted` 通知。

`delay` 默认在 `computation` 调度器上执行，你可以通过参数指定使用其它的调度器。

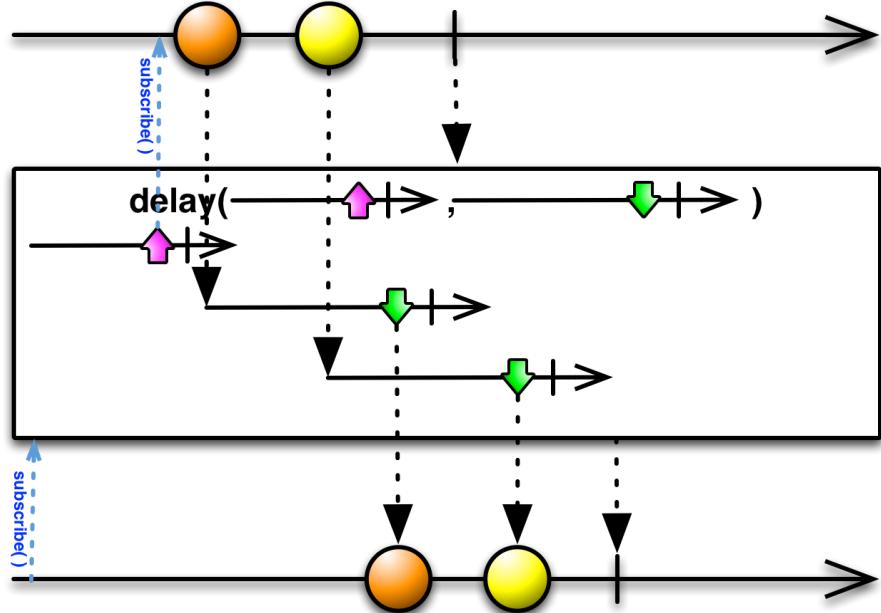
- Javadoc: `delay(long,TimeUnit)`
- Javadoc: `delay()`



另一种 `delay` 不实用常数延时参数，它使用一个函数针对原始Observable的每一项数据返回一个Observable，它监视返回的这个Observable，当任何那样的 Observable 终止时，`delay` 返回的 Observable 就发射关联的那项数据。

这种 `delay` 默认不在任何特定的调度器上执行。

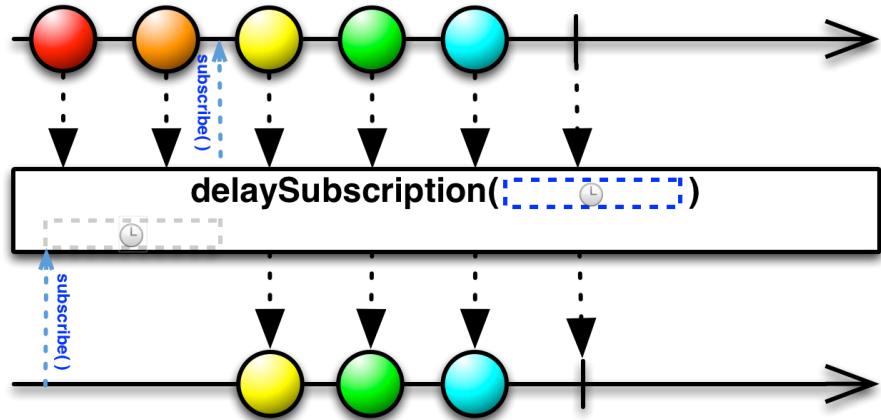
- Javadoc: [delay\(Func1\)](#)



这个版本的 `delay` 对每一项数据使用一个 Observable 作为原始 Observable 的延时定时器。

这种 `delay` 默认不在任何特定的调度器上执行。

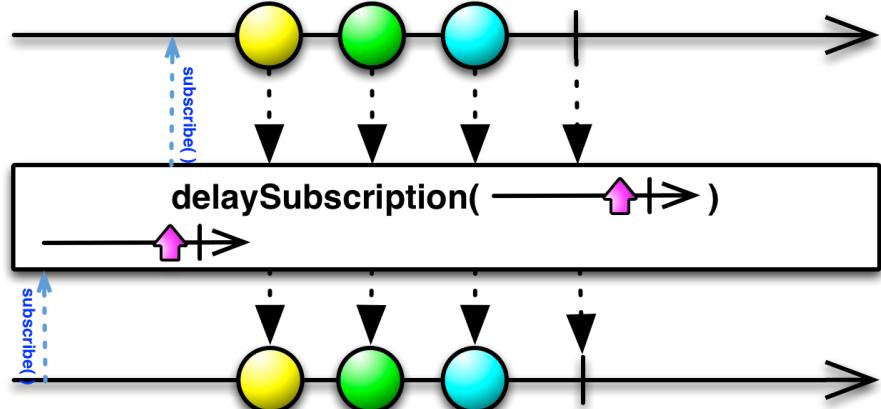
- Javadoc: [delay\(Func0,Func1\)](#)



还有一个操作符 `delaySubscription` 让你可以延迟订阅原始Observable。它结合搜一个定义延时的参数。

`delaySubscription` 默认在 `computation` 调度器上执行，你可以通过参数指定使用其它的调度器。

- Javadoc: `delaySubscription(long,TimeUnit)`
- Javadoc: `delaySubscription(long,TimeUnit,Scheduler)`



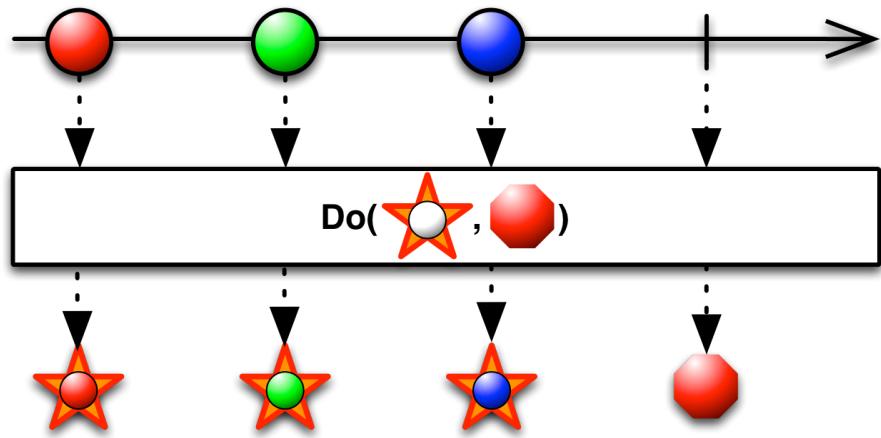
还有一个版本的 `delaySubscription` 使用一个Obseable而不是一个固定的时长来设置订阅延时。

这种 `delaySubscription` 默认不在任何特定的调度器上执行。

- Javadoc: `delaySubscription(Func0)`

Do

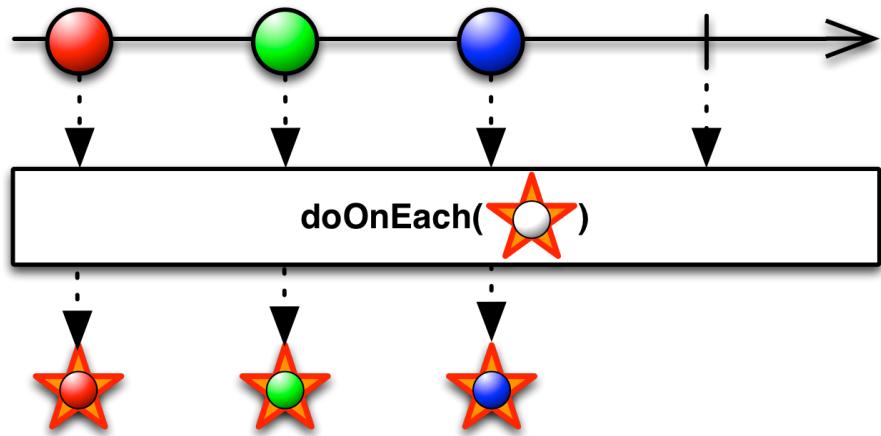
注册一个动作作为原始Observable生命周期事件的一种占位符



你可以注册回调，当Observable的某个事件发生时，Rx会在与Observable链关联的正常通知集合中调用它。Rx实现了多种操作符用于达到这个目的。

RxJava实现了很多 `do` 操作符的变体。

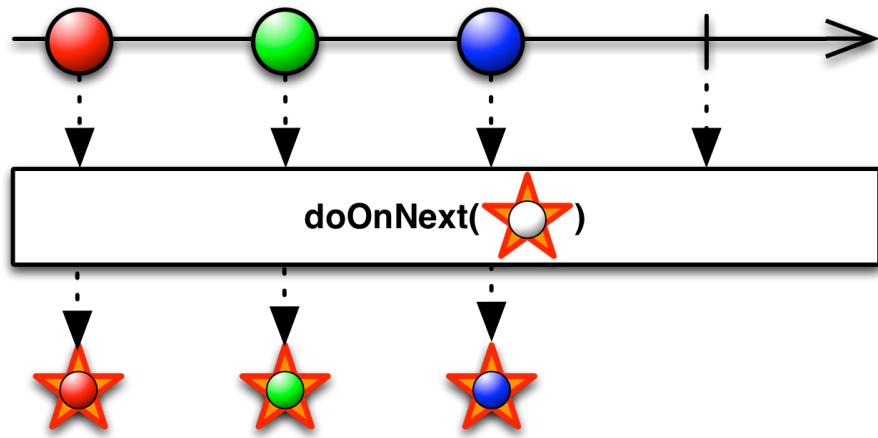
doOnEach



`doOnEach` 操作符让你可以注册一个回调，它产生的Observable每发射一项数据就会调用它一次。你可以以 `Action` 的形式传递参数给它，这个 `Action` 接受一个 `onNext` 的变体 `Notification` 作为它的唯一参数，你也可以传递一个 Observable 给 `doOnEach`，这个 Observable 的 `onNext` 会被调用，就好像它订阅了原始的 Observable 一样。

- Javadoc: `doOnEach(Action1)`
- Javadoc: `doOnEach(Observer)`

doOnNext



`doOnNext` 操作符类似于 `doOnEach(Action1)`，但是它的Action不是接受一个 `Notification` 参数，而是接受发射的数据项。

示例代码

```

Observable.just(1, 2, 3)
    .doOnNext(new Action1<Integer>() {
        @Override
        public void call(Integer item) {
            if( item > 1 ) {
                throw new RuntimeException("Item exceeds
maximum value");
            }
        }
    }).subscribe(new Subscriber<Integer>() {
        @Override
        public void onNext(Integer item) {
            System.out.println("Next: " + item);
        }

        @Override
        public void onError(Throwable error) {
            System.err.println("Error: " +
error.getMessage());
        }

        @Override
        public void onCompleted() {
            System.out.println("Sequence complete.");
        }
    });

```

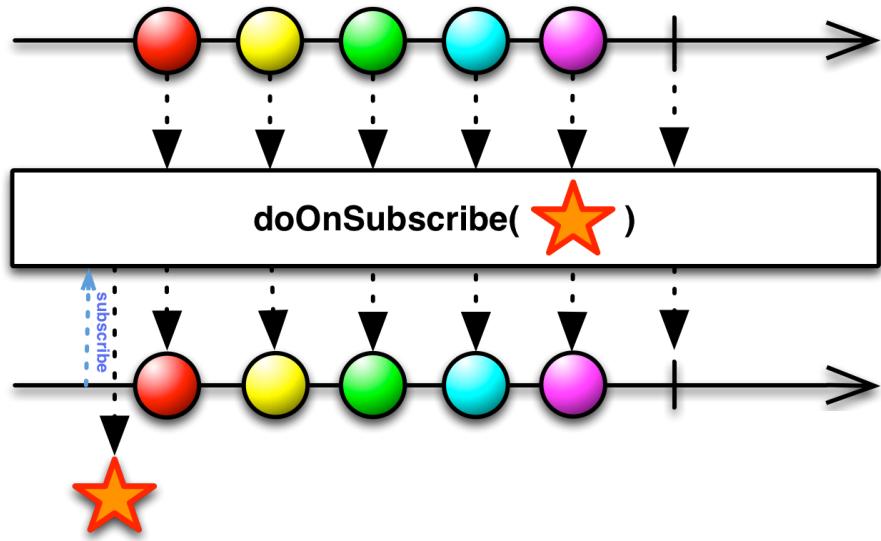
输出

```

Next: 1
Error: Item exceeds maximum value

```

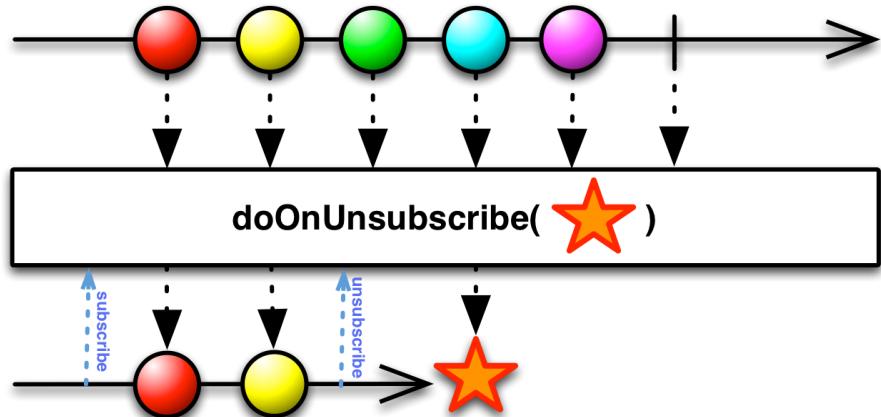
`doOnSubscribe`



`doOnSubscribe` 操作符注册一个动作，当观察者订阅它生成的Observable它就会被调用。

- Javadoc: [doOnSubscribe\(Action0\)](#)

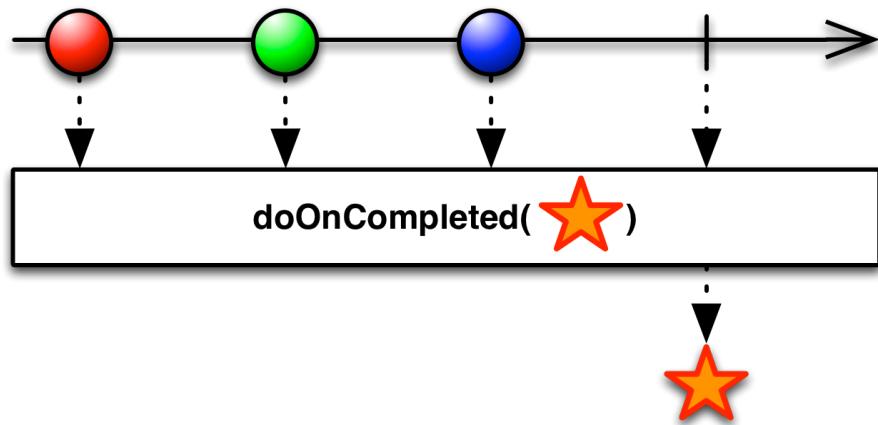
doOnUnsubscribe



`doOnUnsubscribe` 操作符注册一个动作，当观察者取消订阅它生成的 Observable 它就会被调用。

- Javadoc: [doOnUnsubscribe\(Action0\)](#)

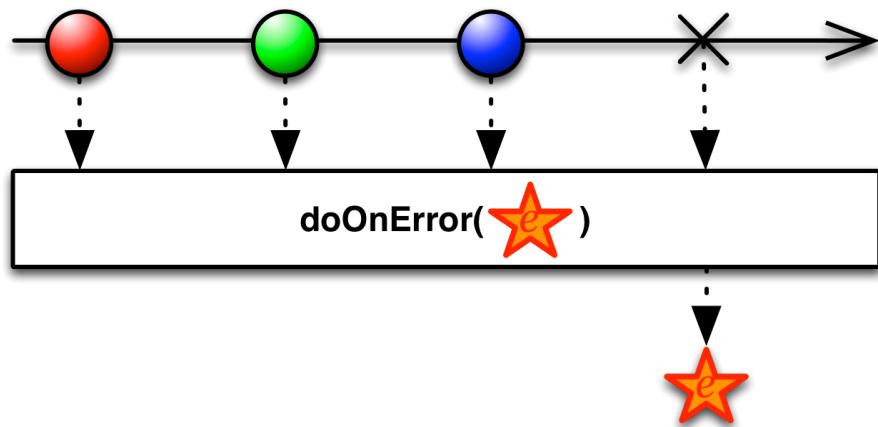
doOnCompleted



`doOnCompleted` 操作符注册一个动作，当它产生的Observable正常终止调用 `onCompleted` 时会被调用。

- Javadoc: [doOnCompleted\(Action0\)](#)

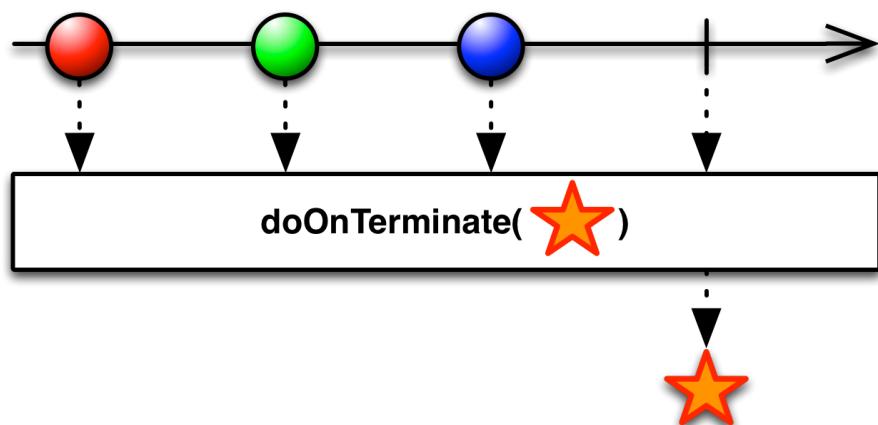
doOnError



`doOnError` 操作符注册一个动作，当它产生的Observable异常终止调用 `onError` 时会被调用。

- Javadoc: [doOnError\(Action0\)](#)

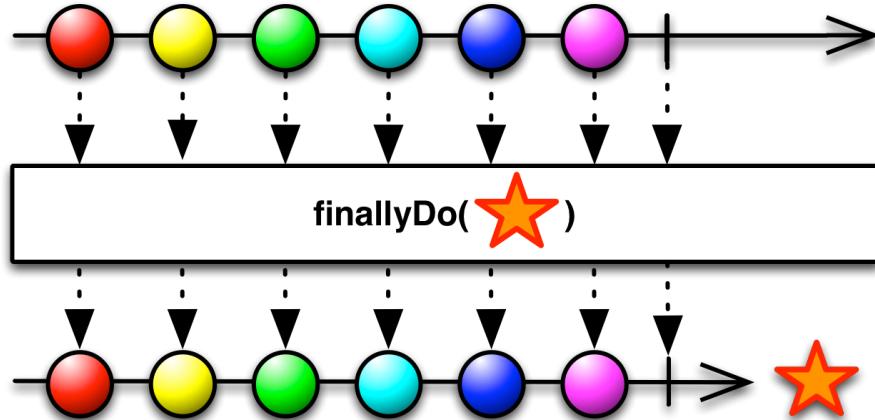
doOnTerminate



`doOnTerminate` 操作符注册一个动作，当它产生的Observable终止之前会被调用，无论是正常还是异常终止。

- Javadoc: `doOnTerminate(Action0)`

finallyDo

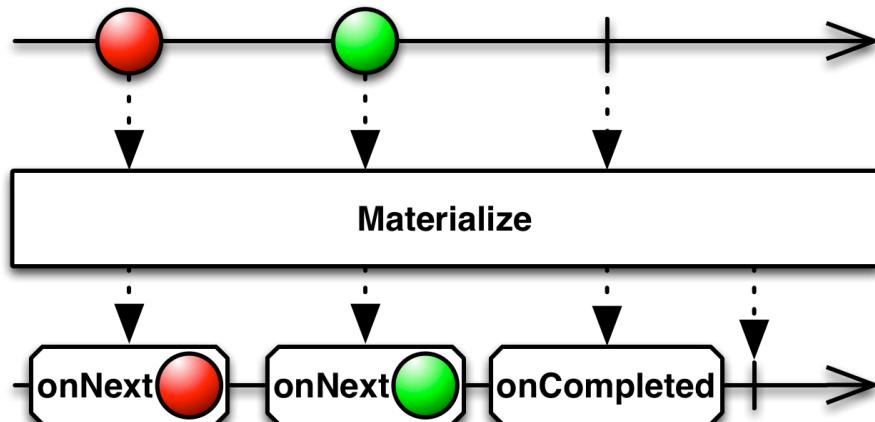


`finallyDo` 操作符注册一个动作，当它产生的Observable终止之后会被调用，无论是正常还是异常终止。

- Javadoc: `finallyDo(Action0)`

Materialize/Dematerialize

`Materialize`将数据项和事件通知都当做数据项发射，`Dematerialize`刚好相反。

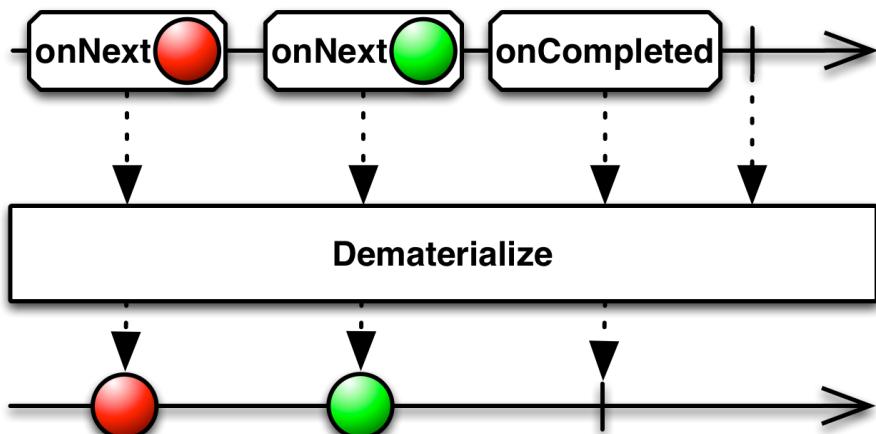


一个合法的有限的Obversable将调用它的观察者的`onNext`方法零次或多次，然后调用观察者的`onCompleted`或`onError`正好一次。`Materialize`操作符将这一系列调用，包括原来的`onNext`通知和终止通知`onCompleted`或`onError`都转换为一个Observable发射的数据序列。

RxJava的`materialize`将来自原始Observable的通知转换为`Notification`对象，然后它返回的Observable会发射这些数据。

`materialize`默认不在任何特定的调度器 (`Scheduler`) 上执行。

- Javadoc: [materialize\(\)](#)



`Dematerialize`操作符是`Materialize`的逆向过程，它将`Materialize`转换的结果还原成它原本的形式。

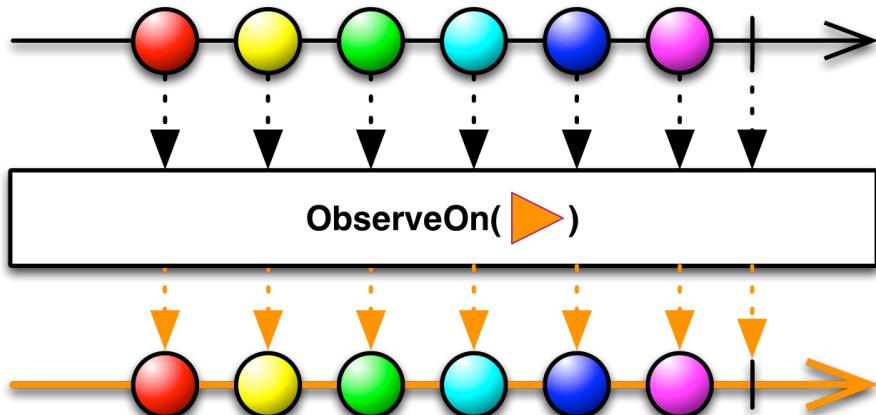
`dematerialize`反转这个过程，将原始Observable发射的Notification对象还原成Observable的通知。

`dematerialize`默认不在任何特定的调度器 (`Scheduler`) 上执行。

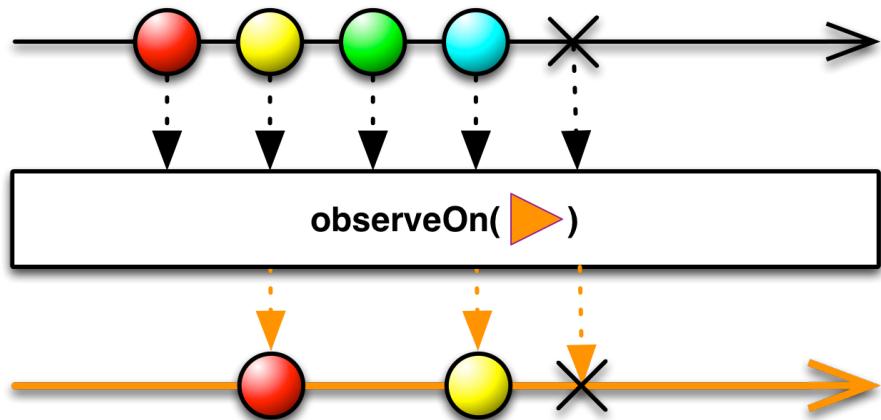
- Javadoc: [dematerialize\(\)](#)

ObserveOn

指定一个观察者在哪个调度器上观察这个Observable



很多ReactiveX实现都使用调度器 "`Scheduler`" 来管理多线程环境中Observable的转场。你可以使用`observeOn`操作符指定Observable在一个特定的调度器上发送通知给观察者 (调用观察者的`onNext`, `onCompleted`, `onError`方法)。



注意：当遇到一个异常时 `observeOn` 会立即向前传递这个 `onError` 终止通知，它不会等待慢速消费的 Observable 接受任何之前它已经收到但还没有发射的数据项。这可能意味着 `onError` 通知会跳到（并吞掉）原始 Observable 发射的数据项前面，正如图例上展示的。

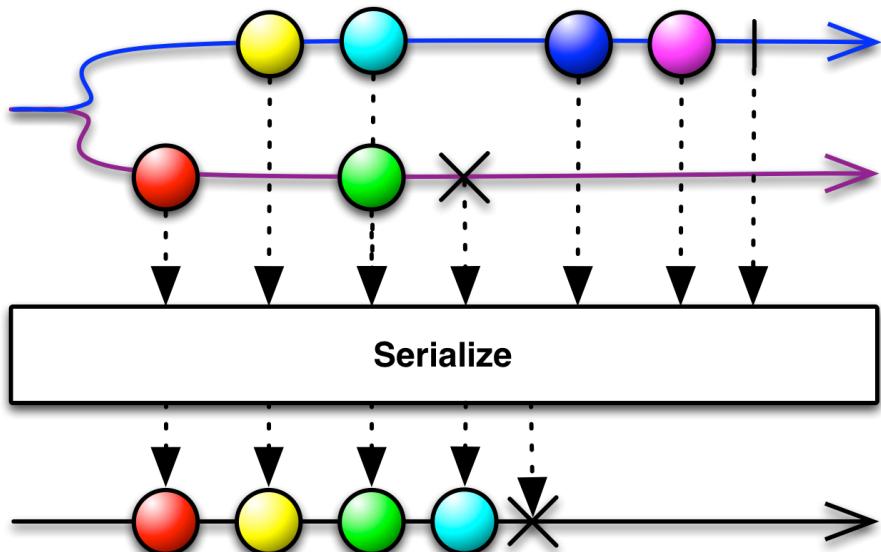
`SubscribeOn` 操作符的作用类似，但它是用于指定 Observable 本身在特定的调度器上执行，它同样会在那个调度器上给观察者发通知。

RxJava 中，要指定 Observable 应该在哪个调度器上调用观察者的 `onNext`, `onCompleted`, `onError` 方法，你需要使用 `observeOn` 操作符，传递给它一个合适的 `Scheduler`。

- Javadoc: `observeOn(Scheduler)`

Serialize

强制一个 Observable 连续调用并保证行为正确



一个 Observable 可以异步调用它的观察者的方法，可能是从不同的线程调用。这可能会让 Observable 行为不正确，它可能会在某一个 `onNext` 调用之前尝试调用 `onCompleted` 或 `onError` 方法，或者从两个不同的线程同时调用 `onNext` 方法。使用 `serialize` 操作符，你可以纠正这个 Observable 的行为，保证它的行为是正确的且是同步的。

RxJava 中的实现是 `serialize`，它默认不在任何特定的调度器上执行。

- Javadoc: `serialize()`

Subscribe

操作来自Observable的发射物和通知

`Subscribe`操作符是连接观察者和Observable的胶水。一个观察者要想看到Observable发射的数据项，或者想要从Observable获取错误和完成通知，它首先必须使用这个操作符订阅那个Observable。

`Subscribe`操作符的一般实现可能会接受一到三个方法（然后由观察者组合它们），或者接受一个实现了包含这三个方法的接口的对象（有时叫做`Observer`或`Subscriber`）：

`onNext`

每当Observable发射了一项数据它就会调用这个方法。这个方法的参数是这个Observable发射的数据项。

`onError`

Observable调用这个方法表示它无法生成期待的数据或者遇到了其它错误。这将停止Observable，它在这之后不会再调用`onNext`或`onCompleted`。`onError`方法的参数是导致这个错误的原因的一个表示（有时可能是一个Exception或Throwable对象，其它时候也可能是一个简单的字符串，取决于具体的实现）。

`onCompleted`

如果没有遇到任何错误，Observable在最后一次调用`onCompleted`之后会调用这个方法。

如果一个Observable直到有一个观察者订阅它才开始发射数据项，就称之为“冷”的Observable；如果一个Observable可能在任何时刻开始发射数据，就称之为“热”的Observable，一个订阅者可能从开始之后的某个时刻开始观察它发射的数据序列，它可能会错过在订阅之前发射的数据。

RxJava中的实现是`subscribe`方法。

如果你使用无参数的版本，它将触发对Observable的一个订阅，但是将忽略它的发射物和通知。这个操作会激活一个“冷”的Observable。

你也可以传递一到三个函数给它，它们会按下面的方法解释：

1. `onNext`
2. `onNext`和`onError`
3. `onNext`, `onError`和`onCompleted`

最后，你还可以传递一个`Observer`或`Subscriber`接口给它，`Observer`接口包含这三个以`on`开头的方法。`Subscriber`接口也实现了这三个方法，而且还添加了几个额外的方法，用于支持使用反压操作(`reactive pull backpressure`)，这让`Subscriber`可以在Observable完成前取消订阅。

`subscribe`方法返回一个实现了`Subscription`接口的对象。这个接口包含`unsubscribe`方法，任何时刻你都可以调用它来断开`subscribe`方法建立的Observable和观察者之间的订阅关系。

- Javadoc: `subscribe()`
- Javadoc: `subscribe(Action1)`
- Javadoc: `subscribe(Action1,Action1)`
- Javadoc: `subscribe(Action1,Action1,Action0)`
- Javadoc: `subscribe(Observer)`
- Javadoc: `subscribe(Subscriber)`

foreach

`forEach`方法是简化版的`subscribe`，你同样可以传递一到三个函数给它，解释和传递给`subscribe`时一样。

不同的是，你无法使用`forEach`返回的对象取消订阅。也没办法传递一个可以用于取消订阅的参数。因此，只有当你明确地需要操作Observable的所有发射物和通知时，你才应该使用这个操作符。

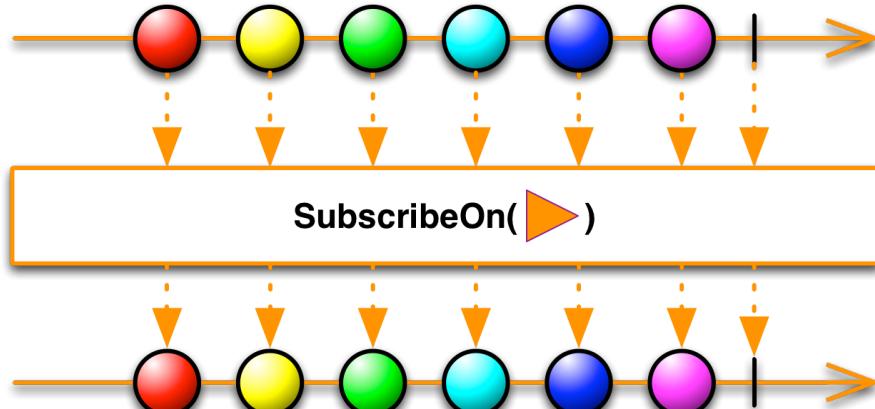
- Javadoc: `forEach(Action1)`
- Javadoc: `forEach(Action1,Action1)`
- Javadoc:
`forEach(Action1,Action1,A/Users/mcxiaoke/github/RxDocs/docs/BlockingObservable.mdction0)`

BlockingObservable

`Blockingobservable`类中也有一个类似的叫作`forEach`的方法。详细的说明见[Blockingobservable](#)

SubscribeOn

指定Observable自身在哪个调度器上执行



很多ReactiveX实现都使用调度器 "[Scheduler](#)" 来管理多线程环境中Observable的转场。你可以使用`subscribeon`操作符指定Observable在一个特定的调度器上运转。

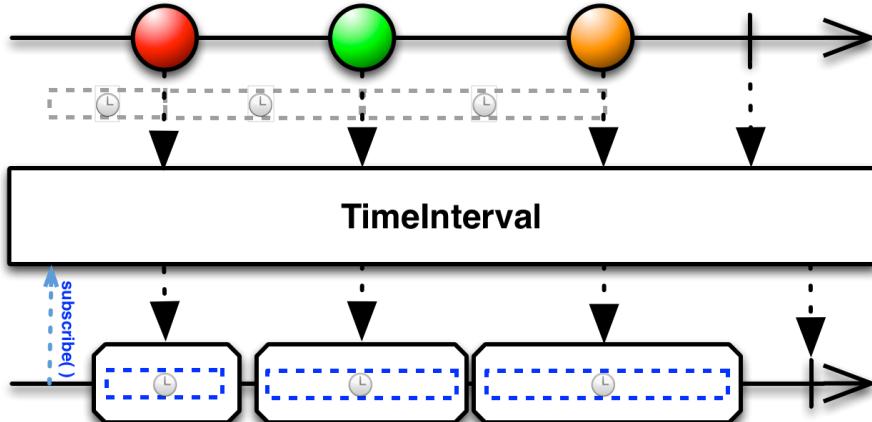
`observeOn`操作符的作用类似，但是功能很有限，它指示Observable在一个指定的调度器上给观察者发通知。

在某些实现中还有一个`unsubscribeOn`操作符。

- Javadoc: `subscribeOn(Scheduler)`
- Javadoc: `unsubscribeOn(Scheduler)`

TimeInterval

将一个发射数据的Observable转换为发射那些数据发射时间间隔的Observable



`TimeInterval`操作符拦截原始Observable发射的数据项，替换为发射表示相邻发射物时间间隔的对象。

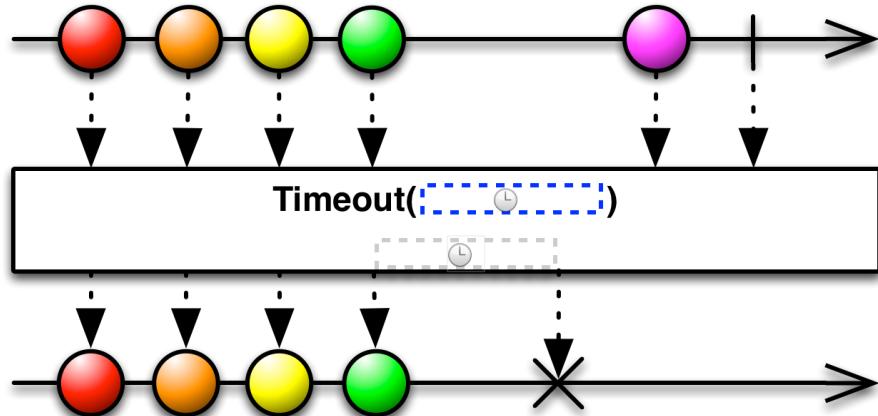
RxJava中的实现为`timeInterval`，这个操作符将原始Observable转换为另一个Observable，后者发射一个标志替换前者的数据项，这个标志表示前者的两个连续发射物之间流逝的时间长度。新的Observable的第一个发射物表示的是在观察者订阅原始Observable到原始Observable发射它的第一项数据之间流逝的时间长度。不存在与原始Observable发射最后一项数据和发射`onCompleted`通知之间时长对应的发射物。

`timeInterval`默认在`immediate`调度器上执行，你可以通过传参数修改。

- Javadoc: `timeInterval()`
- Javadoc: `timeInterval(Scheduler)`

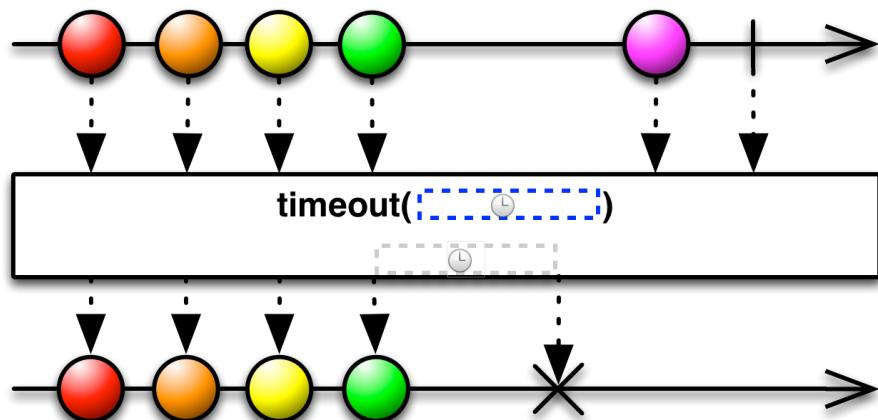
Timeout

对原始Observable的一个镜像，如果过了一个指定的时长仍没有发射数据，它会发一个错误通知



如果原始Observable过了指定的一段时长没有发射任何数据，`Timeout`操作符会以一个`onError`通知终止这个Observable。

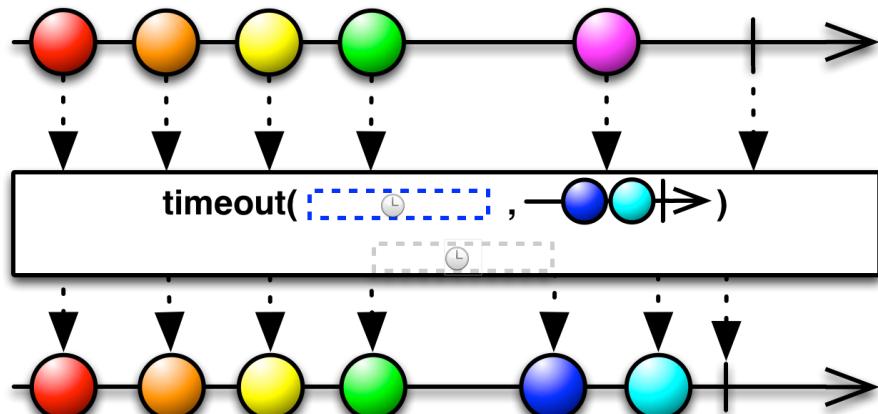
RxJava中的实现为`timeout`，但是有好几个变体。



第一个变体接受一个时长参数，每当原始Observable发射了一项数据，`timeout`就启动一个计时器，如果计时器超过了指定的时长而原始Observable没有发射另一项数据，`timeout`就抛出`TimeoutException`，以一个错误通知终止Observable。

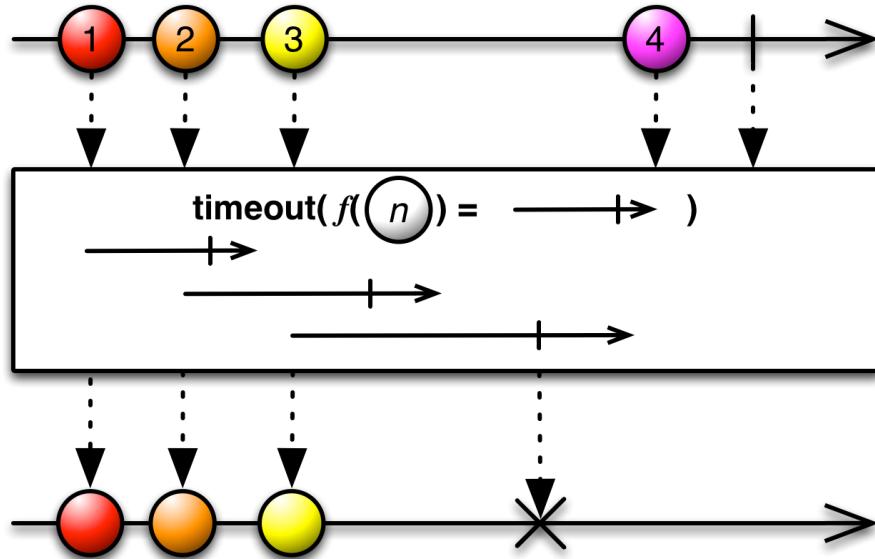
这个`timeout`默认在`computation`调度器上执行，你可以通过参数指定其它的调度器。

- Javadoc: `timeout(long, TimeUnit)`
- Javadoc: `timeout()`



这个版本的 `timeout` 在超时时会切换到使用一个你指定的备用的 Observable，而不是发错误通知。它也默认在 `computation` 调度器上执行。

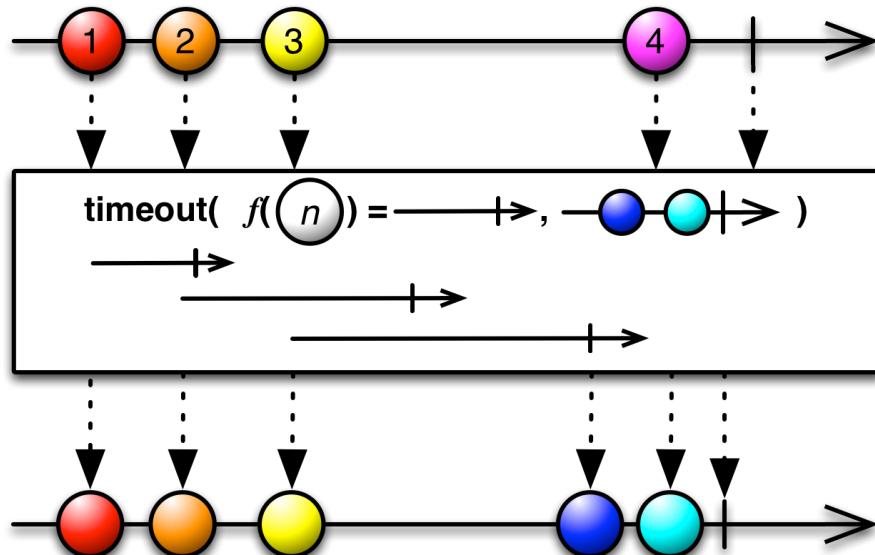
- Javadoc: `timeout(long,TimeUnit,Observable)`
- Javadoc: `timeout(long,TimeUnit,Observable,Scheduler)`



这个版本的 `timeout` 使用一个函数针对原始 Observable 的每一项返回一个 Observable，如果当这个 Observable 终止时原始 Observable 还没有发射另一项数据，就会认为是超时了，`timeout` 就抛出 `TimeoutException`，以一个错误通知终止 Observable。

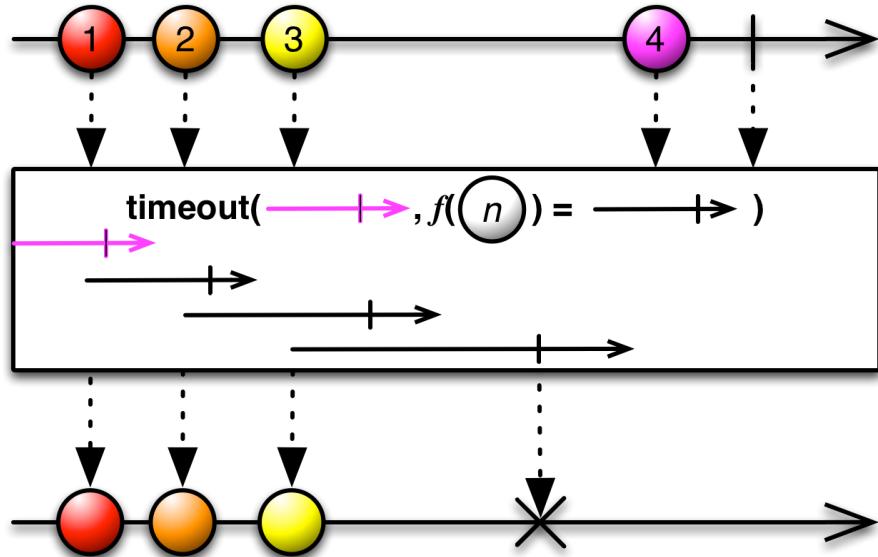
这个 `timeout` 默认在 `immediate` 调度器上执行。

- Javadoc: `timeout(Func1)`



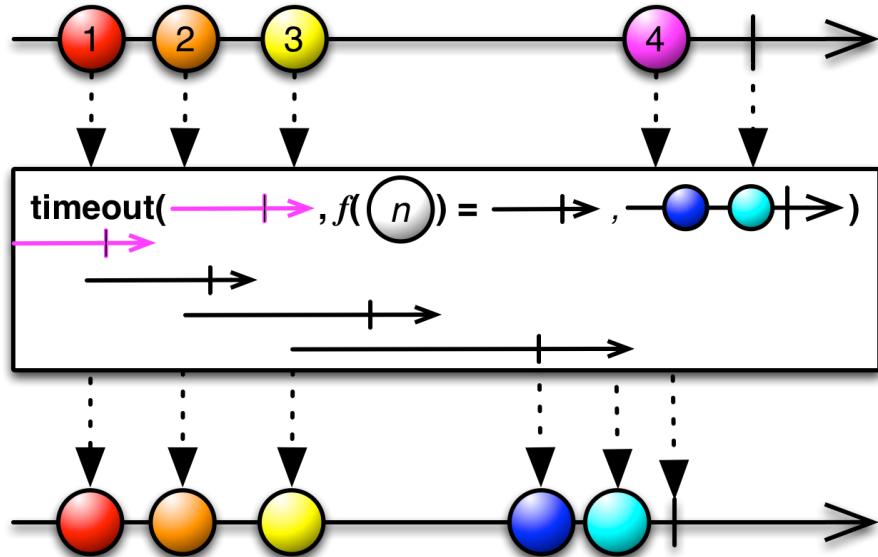
这个版本的 `timeout` 同时指定超时时长和备用的 Observable。它默认在 `immediate` 调度器上执行。

- Javadoc: `timeout(Func1,Observable)`



这个版本的 `time` 除了给每一项设置超时，还可以单独给第一项设置一个超时。
它默认在 `immediate` 调度器上执行。

- Javadoc: `timeout(Func0,Func1)`

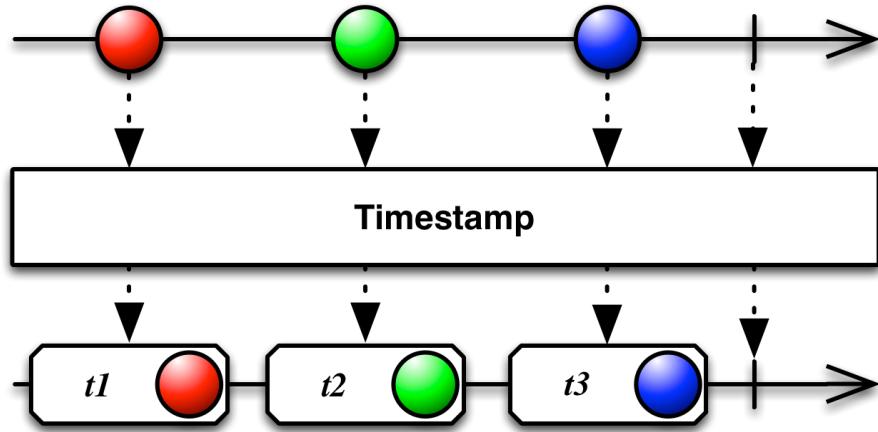


同上，但是同时可以指定一个备用的Observable。它默认在 `immediate` 调度器上执行。

- Javadoc: `timeout(Func0,Func1,Observable)`

Timestamp

给 Observable 发射的数据项附加一个时间戳



RxJava中的实现为 `timestamp`，它将一个发射T类型数据的Observable转换为一个发射类型为 `Timestamped<T>` 的数据的Observable，每一项都包含数据的原始发射时间。

`timestamp` 默认在 `immediate` 调度器上执行，但是可以通过参数指定其它的调度器。

- Javadoc: `timestamp()`
- Javadoc: `timestamp(Scheduler)`

Using

创建一个只在Observable生命周期内存在的一次性资源



`using` 操作符让你可以指示Observable创建一个只在它的生命周期内存在的资源，当Observable终止时这个资源会被自动释放。



`using` 操作符接受三个参数：

1. 一个用户创建一次性资源的工厂函数
2. 一个用于创建Observable的工厂函数
3. 一个用于释放资源的函数

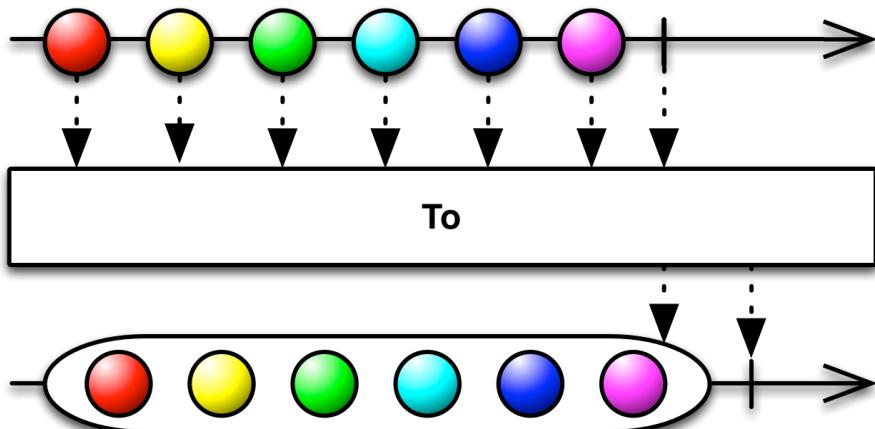
当一个观察者订阅 `using` 返回的 Observable 时，`using` 将会使用 Observable 工厂函数创建观察者要观察的 Observable，同时使用资源工厂函数创建一个你想要创建的资源。当观察者取消订阅这个 Observable 时，或者当观察者终止时（无论是正常终止还是因错误而终止），`using` 使用第三个函数释放它创建的资源。

`using` 默认不在任何特定的调度器上执行。

- Javadoc: [using\(Func0,Func1,Action1\)](#)

To

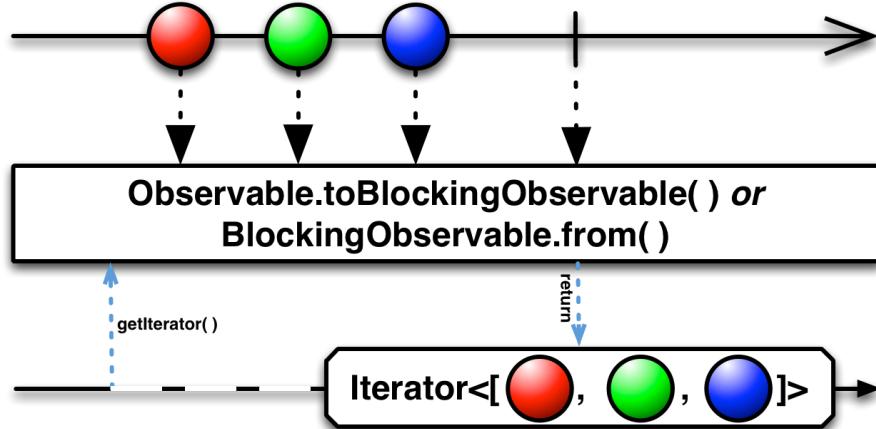
将 Observable 转换为另一个对象或数据结构



ReactiveX 的很多语言特定实现都有一种操作符让你可以将 Observable 或者 Observable 发射的数据序列转换为另一个对象或数据结构。它们中的一些会阻塞直到 Observable 终止，然后生成一个等价的对象或数据结构；另一些返回一个发射那个对象或数据结构的 Observable。

在某些ReactiveX实现中，还有一个操作符用于将Observable转换成阻塞式的。一个阻塞式的Observable在普通的Observable的基础上增加了几个方法，用于操作Observable发射的数据项。

getIterator

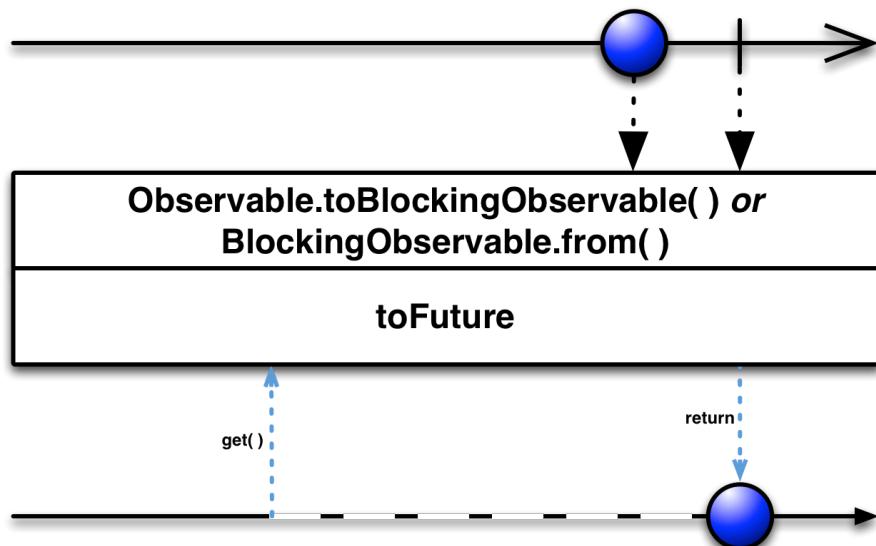


`getIterator`操作符只能用于`BlockingObservable`的子类，要使用它，你首先必须把原始的Observable转换为一个`BlockingObservable`。可以使用这两个操作符：`BlockingObservable.from`或`the observable.toBlocking`。

这个操作符将Observable转换为一个`Iterator`，你可以通过它迭代原始Observable发射的数据集。

- Javadoc: [BlockingObservable.getIterator\(\)](#)

toFuture

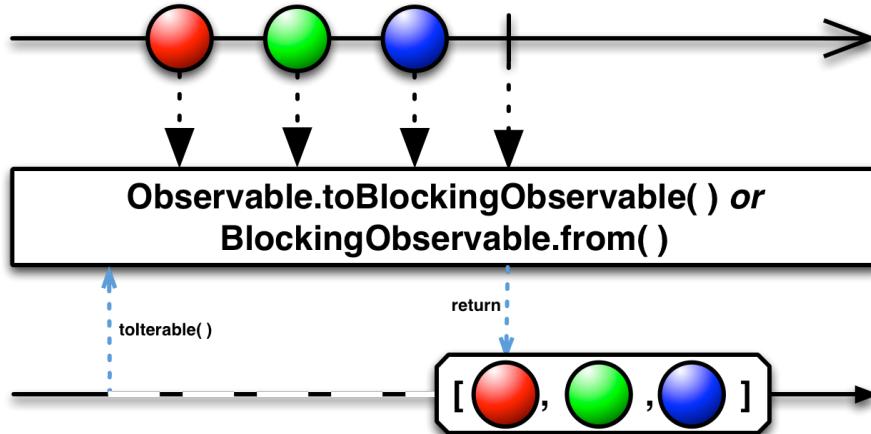


`toFuture`操作符也是只能用于`BlockingObservable`。这个操作符将Observable转换为一个返回单个数据项的`Future`，如果原始Observable发射多个数据项，`Future`会收到一个`IllegalArgumentException`；如果原始Observable没有发射任何数据，`Future`会收到一个`NoSuchElementException`。

如果你想将发射多个数据项的Observable转换为Future，可以这样用：
`myObservable.toList().toBlocking().toFuture()`。

- Javadoc: [BlockingObservable.toFuture\(\)](#)

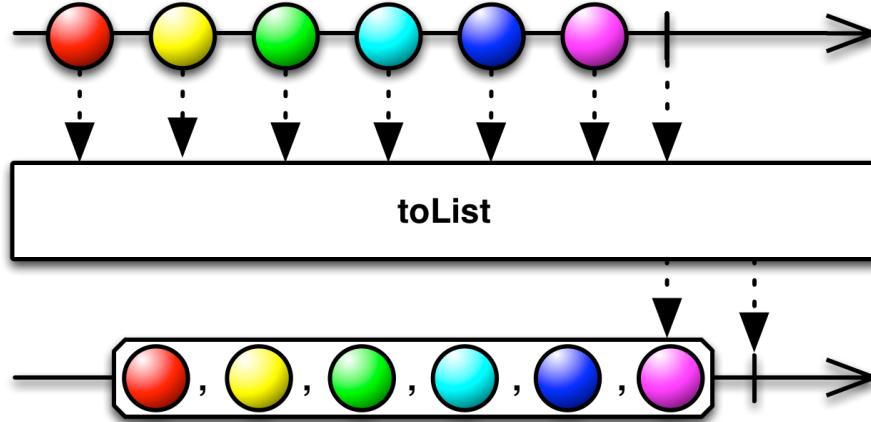
toIterable



`toFuture`操作符也是只能用于`BlockingObservable`。这个操作符将 Observable转换为一个`Iterable`，你可以通过它迭代原始Observable发射的数据集。

- Javadoc: [BlockingObservable.toIterable\(\)](#)

toList



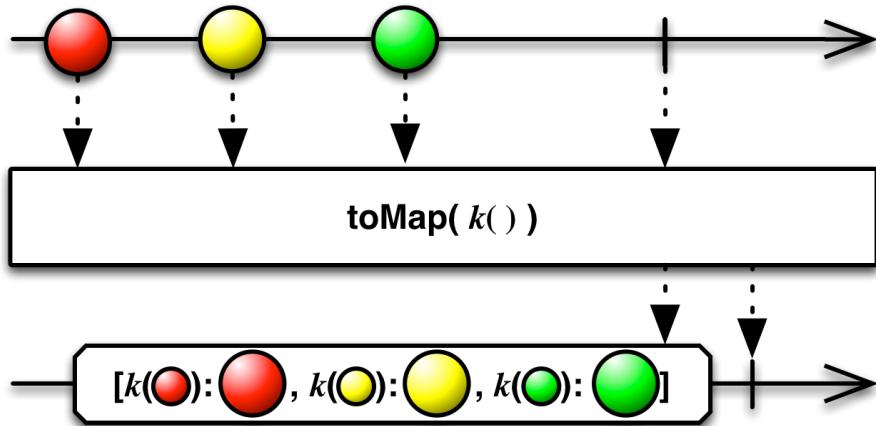
通常，发射多项数据的Observable会为每一项数据调用`onNext`方法。你可以用`toList`操作符改变这个行为，让Observable将多项数据组合成一个`List`，然后调用一次`onNext`方法传递整个列表。

如果原始Observable没有发射任何数据就调用了`onCompleted`，`toList`返回的Observable会在调用`onCompleted`之前发射一个空列表。如果原始Observable调用了`onError`，`toList`返回的Observable会立即调用它的观察者的`onError`方法。

`toList`默认不在任何特定的调度器上执行。

- Javadoc: [toList\(\)](#)

toMap

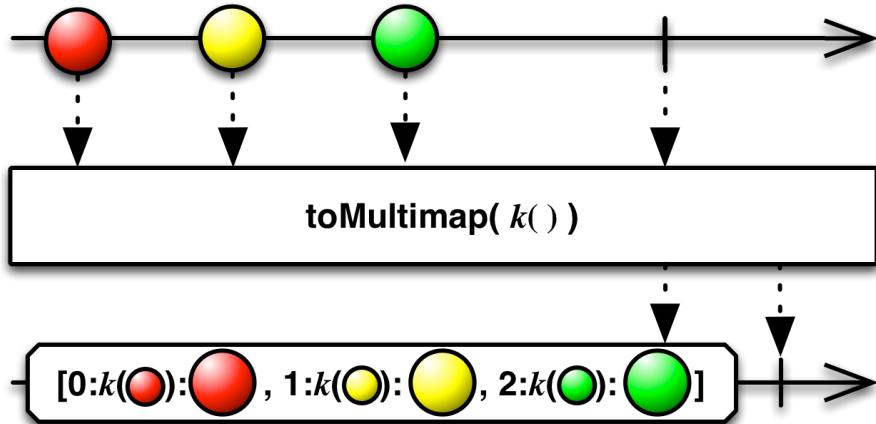


`toMap` 收集原始 Observable 发射的所有数据项到一个 Map（默认是 HashMap）然后发射这个 Map。你可以提供一个用于生成 Map 的 Key 的函数，还可以提供一个函数转换数据项到 Map 存储的值（默认数据项本身就是值）。

`toMap` 默认不在任何特定的调度器上执行。

- Javadoc: [toMap\(Func1\)](#)
- Javadoc: [toMap\(Func1,Func1\)](#)
- Javadoc: [toMap\(Func1,Func1,Func0\)](#)

toMultiMap

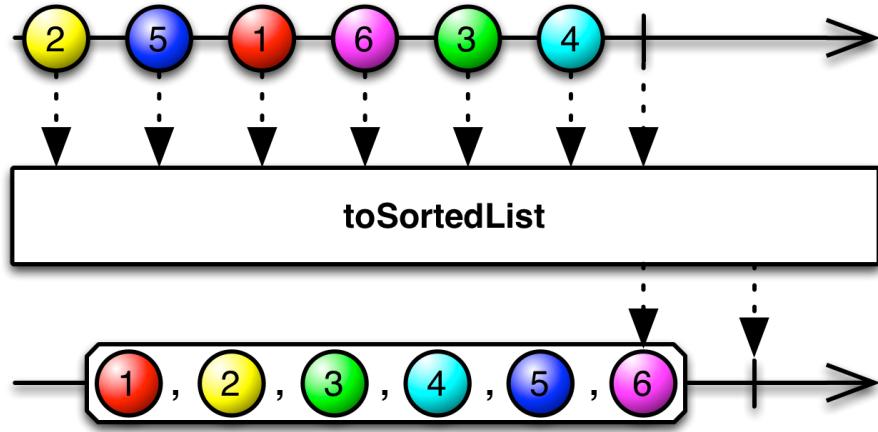


`toMultiMap` 类似于 `toMap`，不同的是，它生成的这个 Map 同时还是一个 `ArrayList`（默认是这样，你可以传递一个可选的工厂方法修改这个行为）。

`toMultiMap` 默认不在任何特定的调度器上执行。

- Javadoc: [toMultiMap\(Func1\)](#)
- Javadoc: [toMultiMap\(Func1,Func1\)](#)
- Javadoc: [toMultiMap\(Func1,Func1,Func0\)](#)
- Javadoc: [toMultiMap\(Func1,Func1,Func0,Func1\)](#)

toSortedList

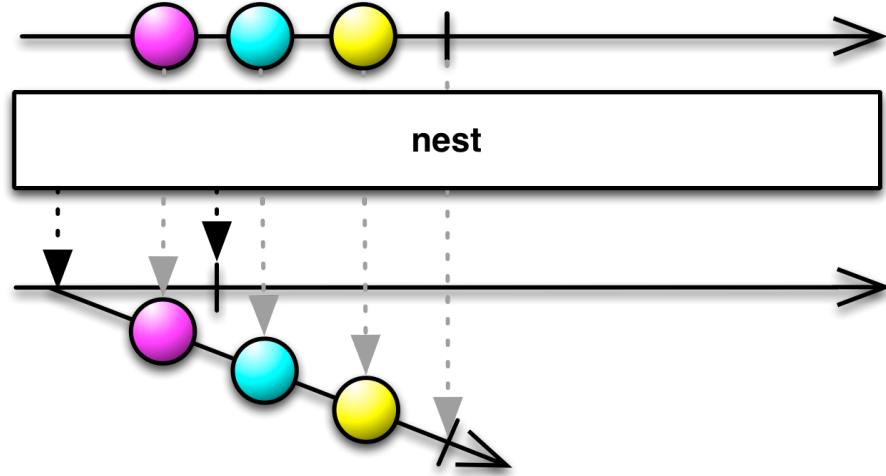


`toSortedList` 类似于 `toList`，不同的是，它会对产生的列表排序，默认是自然升序，如果发射的数据项没有实现 `Comparable` 接口，会抛出一个异常。然而，你也可以传递一个函数作为用于比较两个数据项，这是 `toSortedList` 不会使用 `Comparable` 接口。

`toSortedList` 默认不在任何特定的调度器上执行。

- Javadoc: `toSortedList()`
- Javadoc: `toSortedList(Func2)`

nest



`nest` 操作符有一个特殊的用途：将一个 Observable 转换为一个发射这个 Observable 的 Observable。

条件和布尔操作

这个页面的操作符可用于根据条件发射或变换Observables，或者对它们做布尔运算：

条件操作符

- **amb()** – 给定多个Observable，只让第一个发射数据的Observable发射全部数据
- **defaultIfEmpty()** – 发射来自原始Observable的数据，如果原始Observable没有发射数据，就发射一个默认数据
- (`rxjava-computation-expressions`) **doWhile()** – 发射原始Observable的数据序列，然后重复发射这个序列直到不满足这个条件为止
- (`rxjava-computation-expressions`) **ifThen()** – 只有当某个条件为真时才发射原始Observable的数据序列，否则发射一个空的或默认的序列
- **skipUntil()** – 丢弃原始Observable发射的数据，直到第二个Observable发射了一个数据，然后发射原始Observable的剩余数据
- **skipWhile()** – 丢弃原始Observable发射的数据，直到一个特定的条件为假，然后发射原始Observable剩余的数据
- (`rxjava-computation-expressions`) **switchCase()** – 基于一个计算结果，发射一个指定Observable的数据序列
- **takeUntil()** – 发射来自原始Observable的数据，直到第二个Observable发射了一个数据或一个通知
- **takeWhile() and takeWhileWithIndex()** – 发射原始Observable的数据，直到一个特定的条件为真，然后跳过剩余的数据
- (`rxjava-computation-expressions`) **whileDo()** – 如果条件为 `true`，则发射源Observable数据序列，并且只要条件保持为 `true` 就重复发射此数据序列

(rxjava-computation-expressions) – 表示这个操作符当前是可选包
rxjava-computation-expressions 的一部分，还没有包含在标准
 RxJava的操作符集合里

布尔操作符

- **all()** – 判断是否所有的数据项都满足某个条件
- **contains()** – 判断Observable是否会发射一个指定的值
- **exists() and isEmpty()** – 判断Observable是否发射了一个值
- **sequenceEqual()** – 判断两个Observables发射的序列是否相等

条件和布尔操作

All

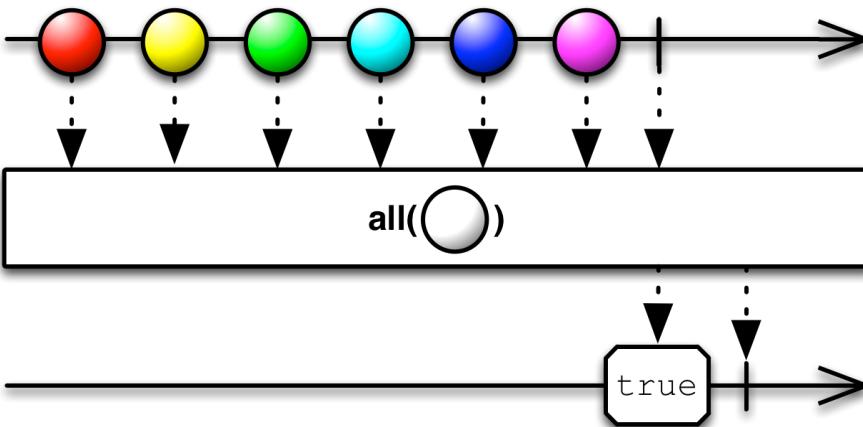
判定是否Observable发射的所有数据都满足某个条件



```
every(x => x < 10)
```



传递一个谓词函数给 `All` 操作符，这个函数接受原始 Observable 发射的数据，根据计算返回一个布尔值。 `All` 返回一个只发射一个单个布尔值的 Observable，如果原始 Observable 正常终止并且每一项数据都满足条件，就返回 `true`；如果原始 Observable 的任何一项数据不满足条件就返回 `false`。



RxJava 将这个操作符实现为 `all`，它默认不在任何特定的调度器上执行。

- Javadoc: `all(Func1)`

Amb

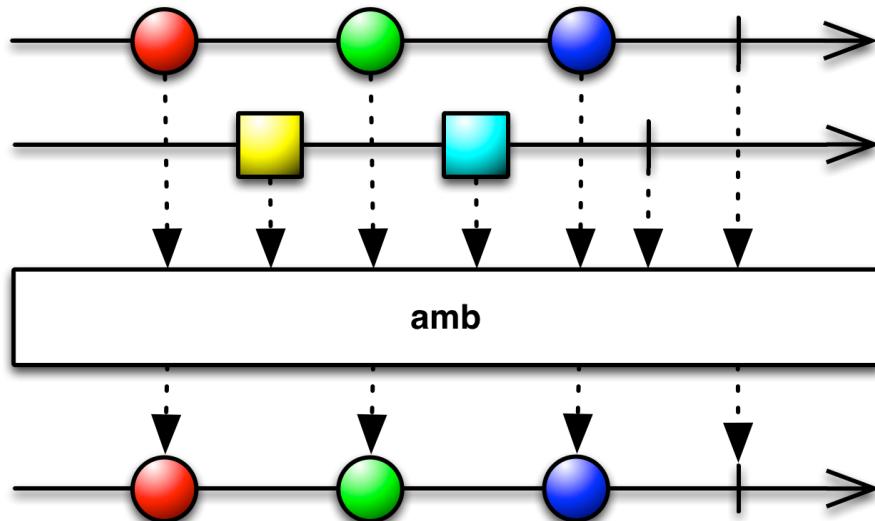
给定两个或多个 Observables，它只发射首先发射数据或通知的那个 Observable 的所有数据



```
amb
```



当你传递多个Observable给**Amb**时，它只发射其中一个Observable的数据和通知：首先发送通知给**Amb**的那个，不管发射的是一项数据还是一个**onError**或**onCompleted**通知。**Amb**将忽略和丢弃其它所有Observables的发射物。

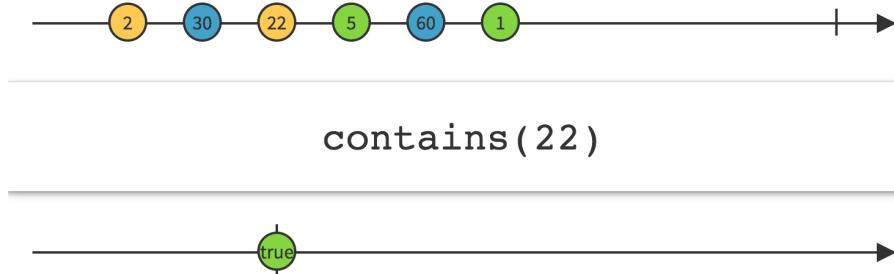


RxJava的实现是**amb**，有一个类似的对象方法**ambwith**。例如，**Observable.amb(o1, o2)** 和 **o1.ambwith(o2)** 是等价的。

这个操作符默认不在任何特定的调度器上执行。

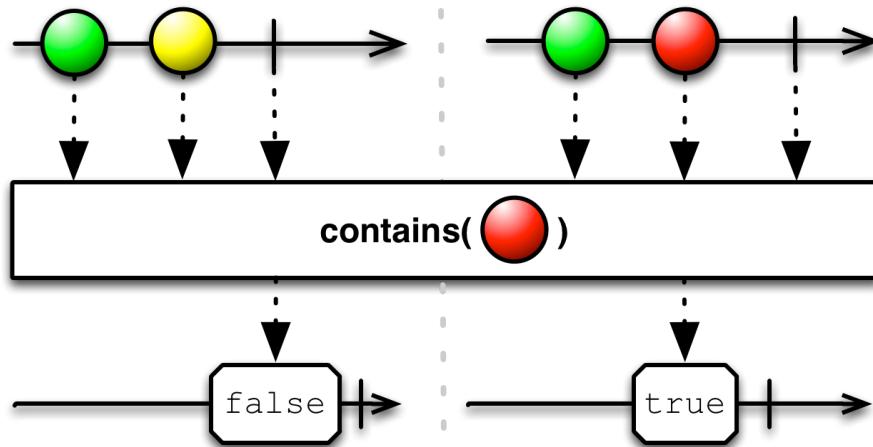
Contains

判定一个Observable是否发射一个特定的值



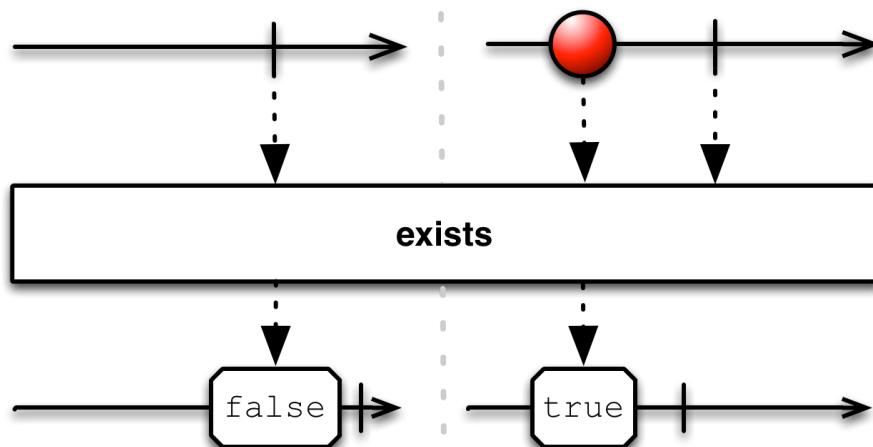
给**Contains**传一个指定的值，如果原始Observable发射了那个值，它返回的Observable将发射**true**，否则发射**false**。

相关的一个操作符**IsEmpty**用于判定原始Observable是否没有发射任何数据。



`contains` 默认不在任何特定的调度器上执行。

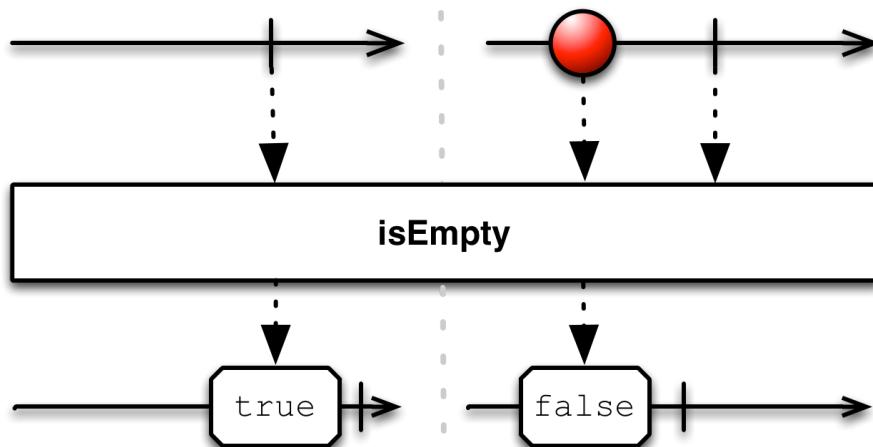
- Javadoc: `contains(Object)`



RxJava中还有一个`exists`操作符，它通过一个谓词函数测试原始Observable发射的数据，只要任何一项满足条件就返回一个发射true的Observable，否则返回一个发射false的Observable。

`exists` 默认不在任何特定的调度器上执行。

- Javadoc: `exists(Func1)`

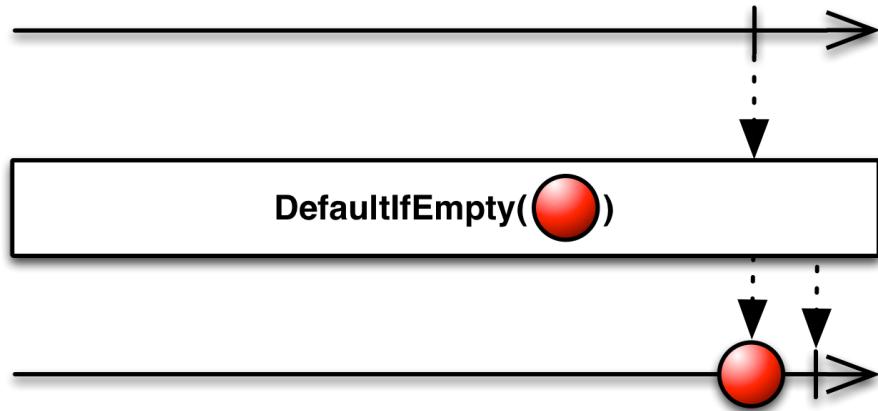


`isEmpty` 默认不在任何特定的调度器上执行。

- Javadoc: `isEmpty()`

DefaultIfEmpty

发射来自原始Observable的值，如果原始Observable没有发射任何值，就发射一个默认值



`DefaultIfEmpty`简单的精确地发射原始Observable的值，如果原始Observable没有发射任何数据正常终止（以`onCompleted`的形式），`DefaultIfEmpty`返回的Observable就发射一个你提供的默认值。

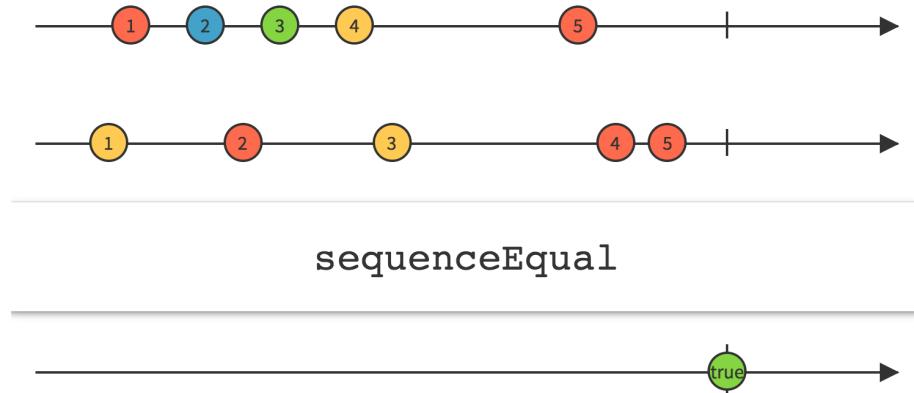
RxJava将这个操作符实现为`defaultIfEmpty`。它默认不在任何特定的调度器上执行。

- Javadoc: [defaultIfEmpty\(T\)](#)

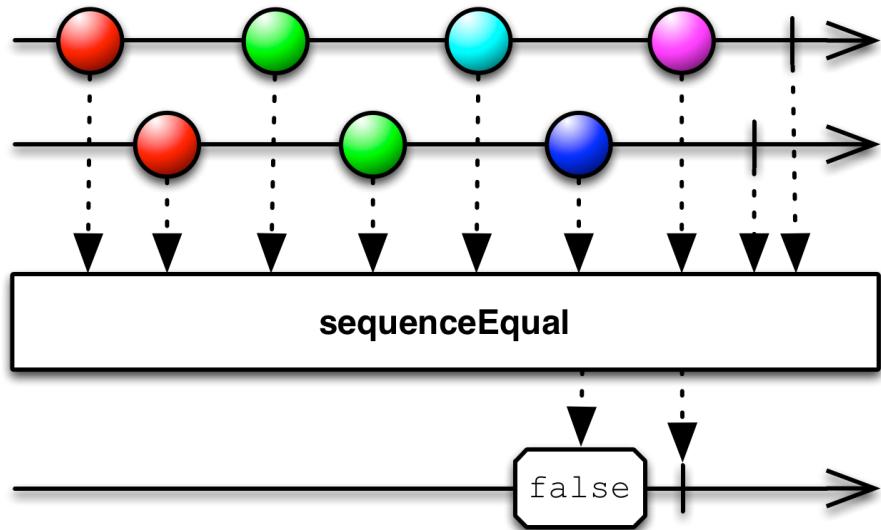
还有一个新的操作符`switchIfEmpty`，不在RxJava 1.0.0版中，它和`defaultIfEmpty`类似，不同的是，如果原始Observable没有发射数据，它发射一个备用Observable的发射物。

SequenceEqual

判定两个Observables是否发射相同的数据序列。



传递两个Observable给`sequenceEqual`操作符，它会比较两个Observable的发射物，如果两个序列是相同的（相同的数据，相同的顺序，相同的终止状态），它就发射`true`，否则发射`false`。



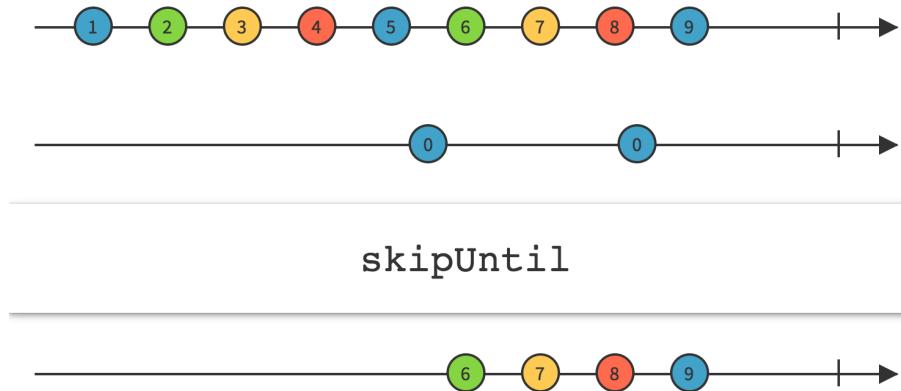
它还有一个版本接受第三个参数，可以传递一个函数用于比较两个数据项是否相同。

这个操作符默认不在任何特定的调度器上执行。

- Javadoc: [sequenceEqual\(Observable, Observable\)](#)
- Javadoc: [sequenceEqual\(Observable, Observable, Func2\)](#)

SkipUntil

丢弃原始Observable发射的数据，直到第二个Observable发射了一项数据



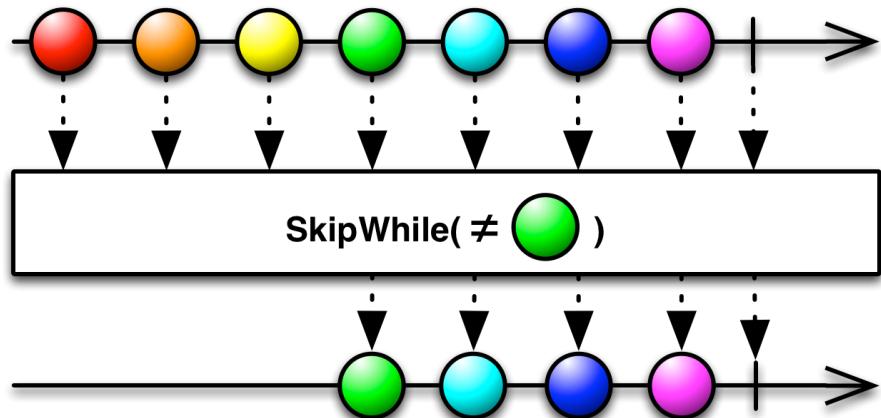
`skipUntil` 订阅原始的Observable，但是忽略它的发射物，直到第二个Observable发射了一项数据那一刻，它开始发射原始Observable。

RxJava中对应的是`skipUntil`，它默认不在任何特定的调度器上执行。

- Javadoc: [skipUntil\(Observable\)](#)

SkipWhile

丢弃Observable发射的数据，直到一个指定的条件不成立



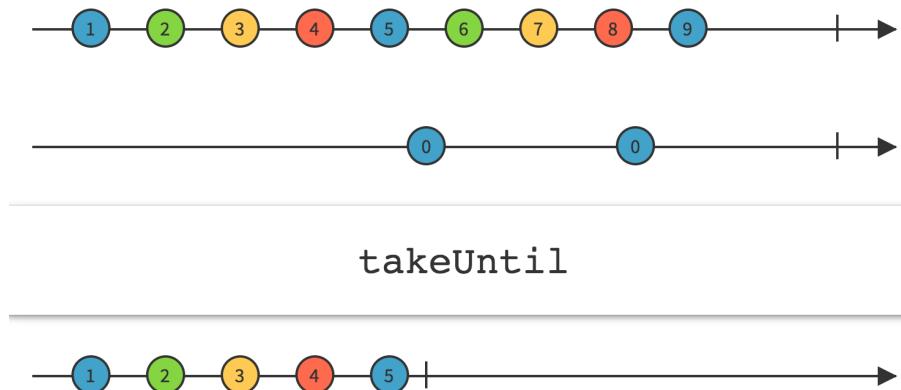
`Skipwhile` 订阅原始的Observable，但是忽略它的发射物，直到你指定的某个条件变为false的那一刻，它开始发射原始Observable。

`skipwhile` 默认不在任何特定的调度器上执行。

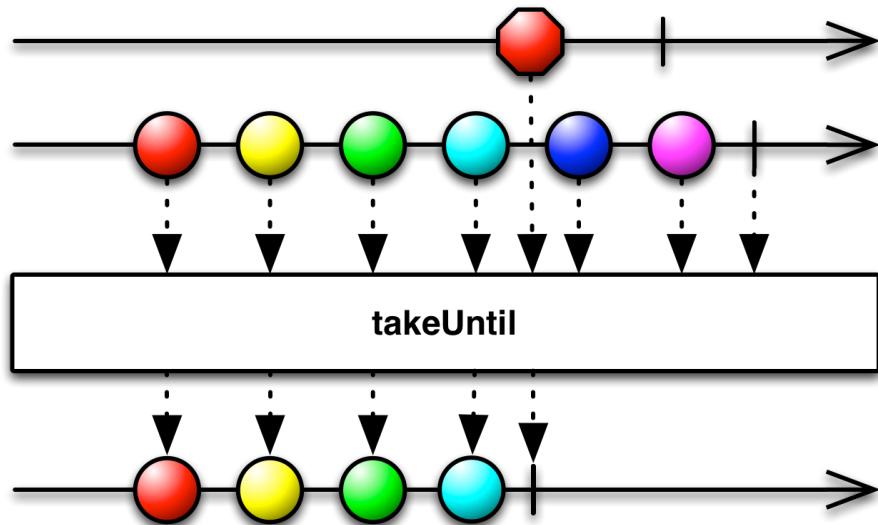
- Javadoc: [skipWhile\(Func1\)](#)

TakeUntil

当第二个Observable发射了一项数据或者终止时，丢弃原始Observable发射的任何数据



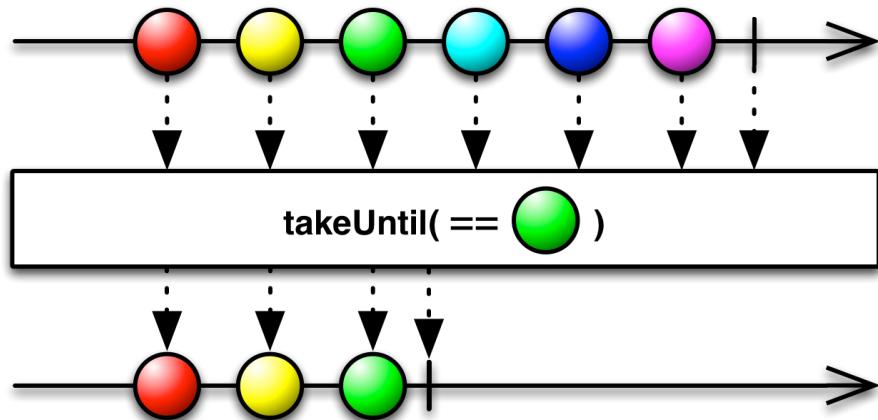
`TakeUntil` 订阅并开始发射原始Observable，它还监视你提供的第二个Observable。如果第二个Observable发射了一项数据或者发射了一个终止通知，`TakeUntil` 返回的Observable会停止发射原始Observable并终止。



RxJava中的实现是 `takeUntil`。注意：第二个Observable发射一项数据或一个 `onError` 通知或一个 `onCompleted` 通知都会导致 `takeUntil` 停止发射数据。

`takeUntil` 默认不在任何特定的调度器上执行。

- Javadoc: [takeUntil\(Observable\)](#)

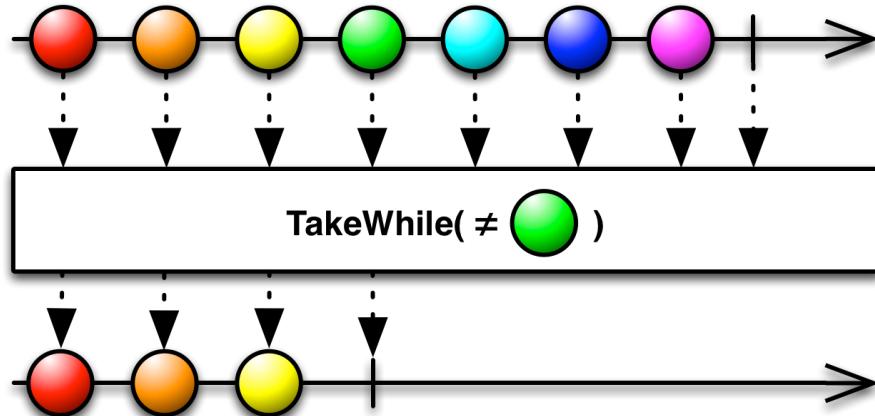


还有一个版本的 `takeUntil`，不在RxJava 1.0.0版中，它使用一个谓词函数而不是第二个Observable来判定是否需要终止发射数据，它的行为类似于 `takewhile`。

- Javadoc: [takeUntil\(Func1\)](#)

TakeWhile

发射Observable发射的数据，直到一个指定的条件不成立



`Takewhile` 发射原始Observable，直到你指定的某个条件不成立的那一刻，它停止发射原始Observable，并终止自己的Observable。

RxJava中的`takewhile`操作符返回一个镜像原始Observable行为的Observable，直到某一项数据你指定的函数返回`false`那一刻，这个新的Observable发射`onCompleted`终止通知。

`takewhile`默认不在任何特定的调度器上执行。

- Javadoc: [takeWhile\(Func1\)](#)

算术和聚合操作

本页展示的操作符用于对整个序列执行算法操作或其它操作，由于这些操作必须等待数据发射完成（通常也必须缓存这些数据），它们对于非常长或者无限的序列来说是危险的，不推荐使用。

`rxjava-math` 模块的操作符

- `averageInteger()` – 求序列平均数并发射
- `averageLong()` – 求序列平均数并发射
- `averageFloat()` – 求序列平均数并发射
- `averageDouble()` – 求序列平均数并发射
- `max()` – 求序列最大值并发射
- `maxBy()` – 求最大key对应的值并发射
- `min()` – 求最小值并发射
- `minBy()` – 求最小Key对应的值并发射
- `sumInteger()` – 求和并发射
- `sumLong()` – 求和并发射
- `sumFloat()` – 求和并发射
- `sumDouble()` – 求和并发射

其它聚合操作符

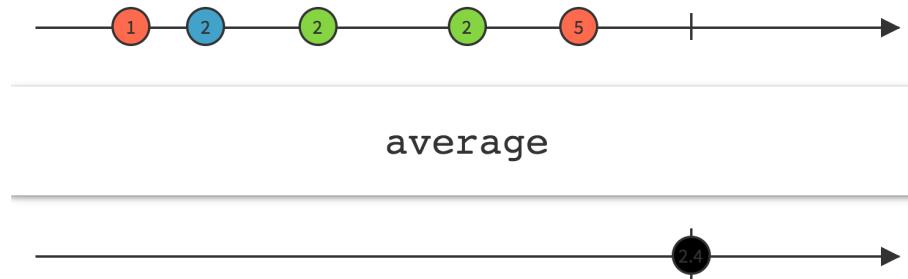
- `concat()` – 顺序连接多个Observables
- `count()` and `countLong()` – 计算数据项的个数并发射结果

- **reduce()** – 对序列使用reduce()函数并发射最终的结果
- **collect()** – 将原始Observable发射的数据放到一个单一的可变的数据结构中，然后返回一个发射这个数据结构的Observable
- **toList()** – 收集原始Observable发射的所有数据到一个列表，然后返回这个列表
- **toSortedList()** – 收集原始Observable发射的所有数据到一个有序列表，然后返回这个列表
- **toMap()** – 将序列数据转换为一个Map，Map的key是根据一个函数计算的
- **toMultiMap()** – 将序列数据转换为一个列表，同时也是一个Map，Map的key是根据一个函数计算的

算术和聚合操作

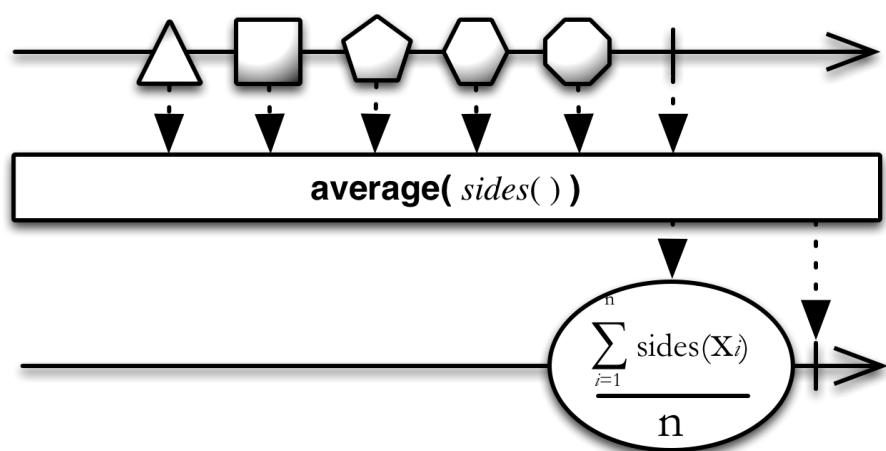
Average

计算原始Observable发射数字的平均值并发射它



Average 操作符操作符一个发射数字的Observable，并发射单个值：原始 Observable发射的数字序列的平均值。

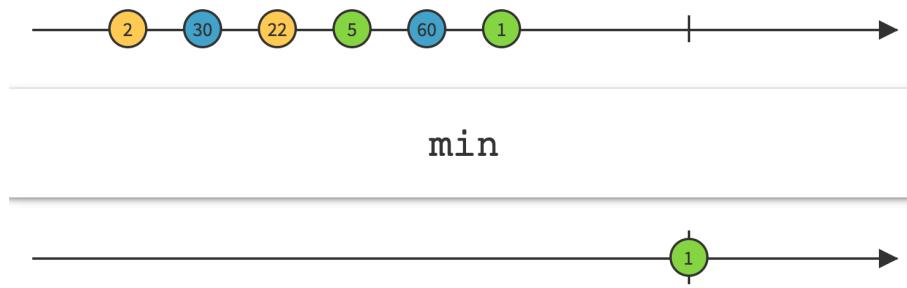
这个操作符不包含在RxJava核心模块中，它属于不同的 `rxjava-math` 模块。它被实现为四个操作符：`averageDouble`, `averageFloat`, `averageInteger`, `averageLong`。



如果原始Observable不发射任何数据，这个操作符会抛异常：
`IllegalArgumentException`。

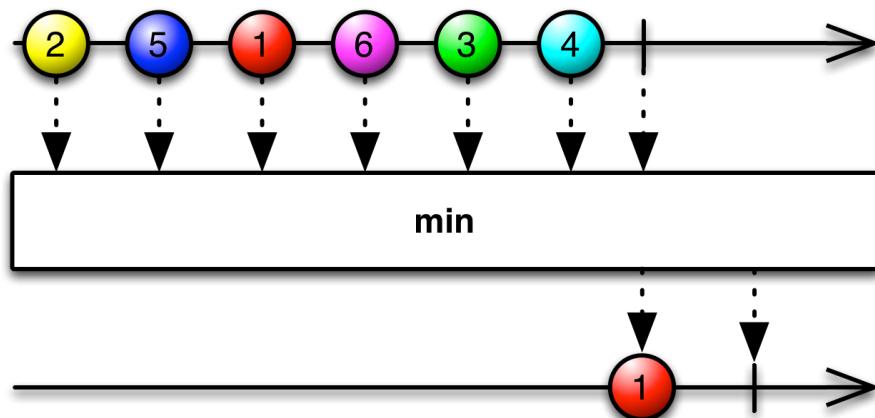
Min

发射原始Observable的最小值

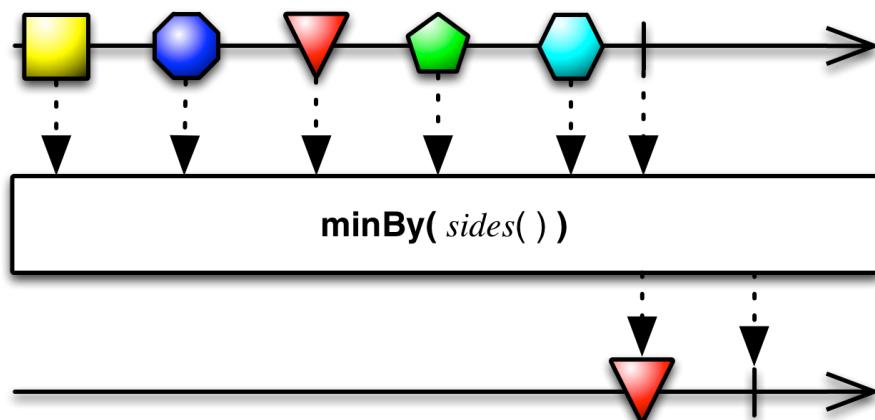


`min`操作符操作一个发射数值的Observable并发射单个值：最小的那个值。

RxJava中，`min`属于`rxjava-math`模块。



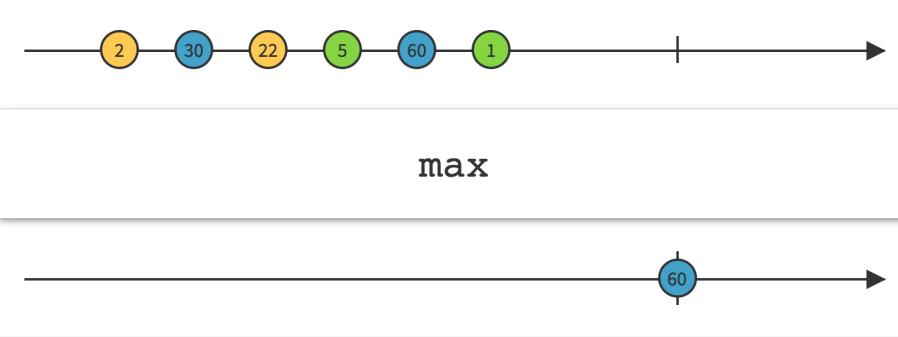
`min`接受一个可选参数，用于比较两项数据的大小，如果最小值的数据超过一项，`min`会发射原始Observable最近发射的那一项。



`minBy`类似于`min`，但是它发射的不是最小值，而是发射Key最小的项，Key由你指定的一个函数生成。

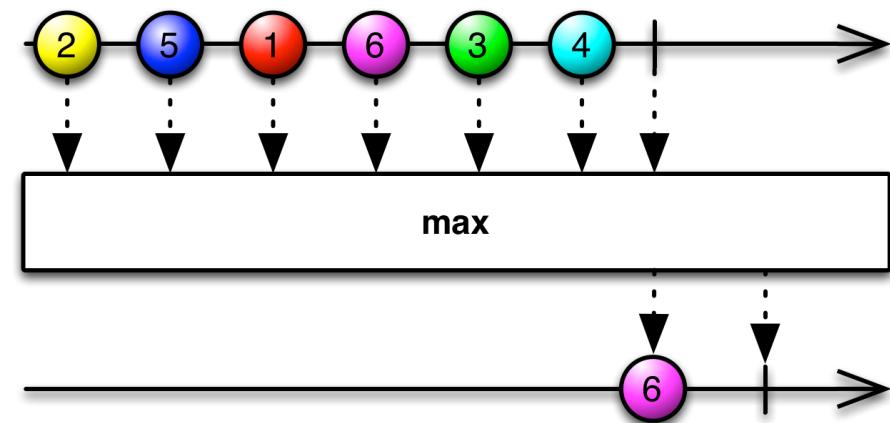
Max

发射原始Observable的最大值

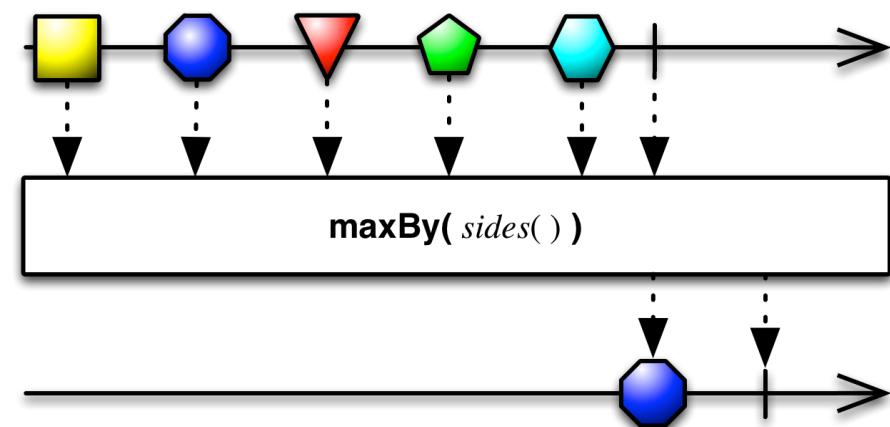


`Max` 操作符操作一个发射数值的 Observable 并发射单个值：最大的那个值。

RxJava中，`max` 属于 `rxjava-math` 模块。



`max` 接受一个可选参数，用于比较两项数据的大小，如果最大值的数据超过一项，`max` 会发射原始 Observable 最近发射的那一项。



`maxBy` 类似于 `max`，但是它发射的不是最大值，而是发射 Key 最大的项，Key 由你指定的一个函数生成。

Count

计算原始 Observable 发射物的数量，然后只发射这个值



```
count(x => x > 10)
```



Count操作符将一个Observable转换成一个发射单个值的Observable，这个值表示原始Observable发射的数据的数量。

如果原始Observable发生错误终止，**Count**不发射数据而是直接传递错误通知。
如果原始Observable永远不终止，**Count**既不会发射数据也不会终止。

RxJava的实现是**count**和**countLong**。

示例代码

```
String[] items = new String[] { "one", "two", "three" };
assertEquals( new Integer(3),
Observable.from(items).count().toBlocking().single() );
```

- Javadoc: [count\(\)](#)
- Javadoc: [countLong\(\)](#)

Sum

计算Observable发射的数值的和并发射这个和

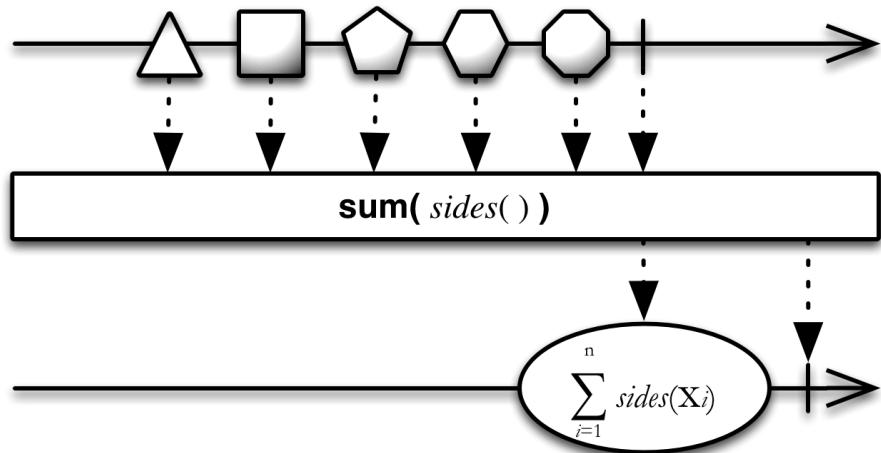


```
sum
```



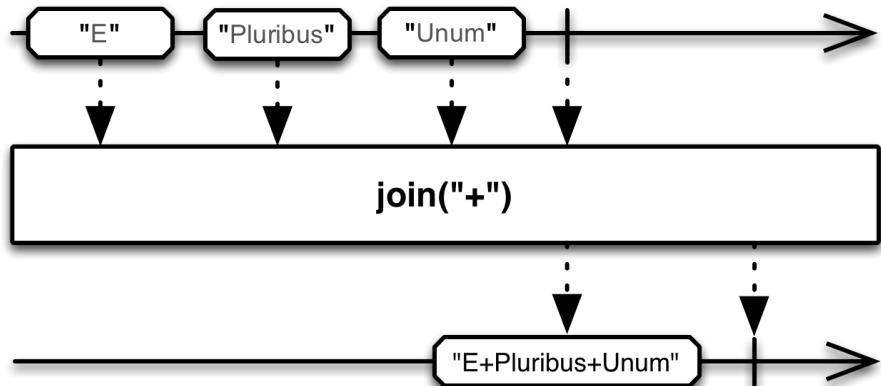
Sum操作符操作一个发射数值的Observable，仅发射单个值：原始Observable所有数值的和。

RxJava的实现是**sumDouble**, **sumFloat**, **sumInteger**, **sumLong**，它们不是RxJava核心模块的一部分，属于**rxjava-math**模块。



你可以使用一个函数，计算Observable每一项数据的函数返回值的和。

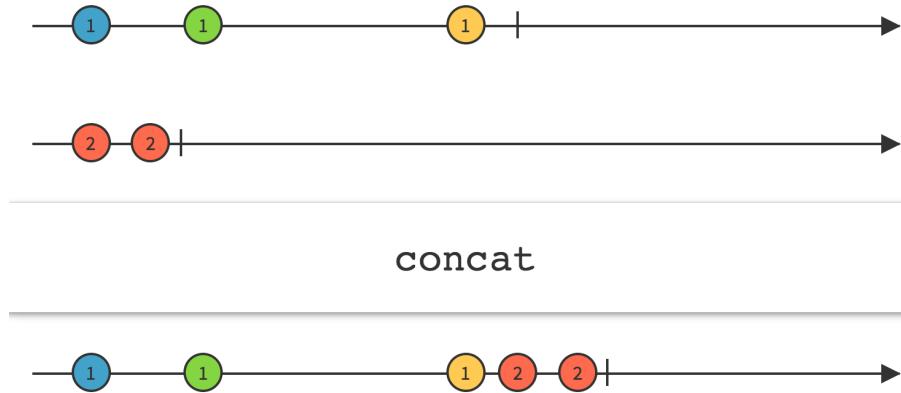
在 `StringObservable` 类（这个类不是RxJava核心模块的一部分）中有一个 `stringConcat` 操作符，它将一个发射字符串序列的 Observable 转换为一个发射单个字符串的 Observable，后者这个字符串表示的是前者所有字符串的连接。



`StringObservable` 类还有一个 `join` 操作符，它将一个发射字符串序列的 Observable 转换为一个发射单个字符串的 Observable，后者这个字符串表示的是前者所有字符串以你指定的分界符连接的结果。

Concat

不交错的发射两个或多个 Observable 的发射物



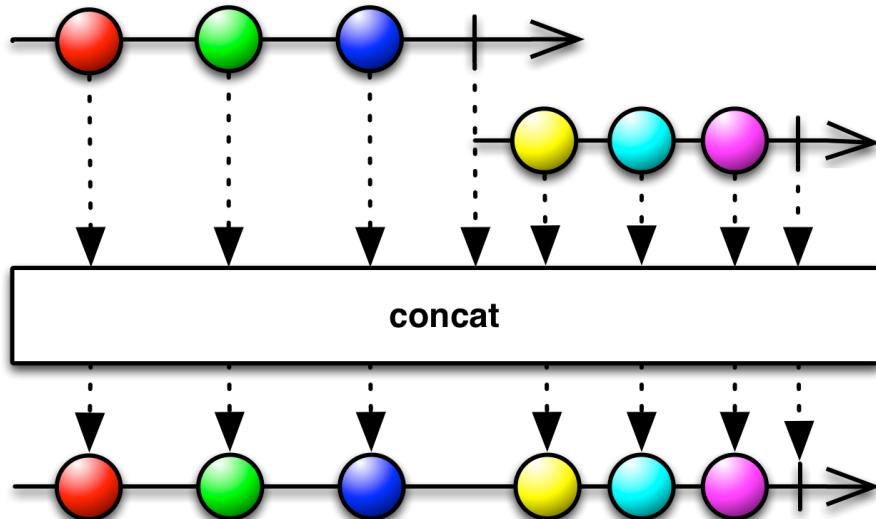
`Concat`操作符连接多个Observable的输出，就好像它们是一个Observable，第一个Observable发射的所有数据在第二个Observable发射的任何数据前面，以此类推。

直到前面一个Observable终止，`Concat`才会订阅额外的一个Observable。注意：因此，如果你尝试连接一个“热”Observable（这种Observable在创建后立即开始发射数据，即使没有订阅者），`concat`将不会看到也不会发射它之前发射的任何数据。

在ReactiveX的某些实现中有一种`ConcatMap`操作符（名字可能叫`concat_all`, `concat_map`, `concatMapObserver`, `for`, `forIn/for_in`, `mapcat`, `selectConcat`或`selectConcatObserver`），他会变换原始Observable发射的数据到一个对应的Observable，然后再按观察和变换的顺序进行连接操作。

`Startwith`操作符类似于`Concat`，但是它是插入到前面，而不是追加那些Observable的数据到原始Observable发射的数据序列。

`Merge`操作符也差不多，它结合两个或多个Observable的发射物，但是数据可能交错，而`Concat`不会让多个Observable的发射物交错。



RxJava中的实现叫`concat`。

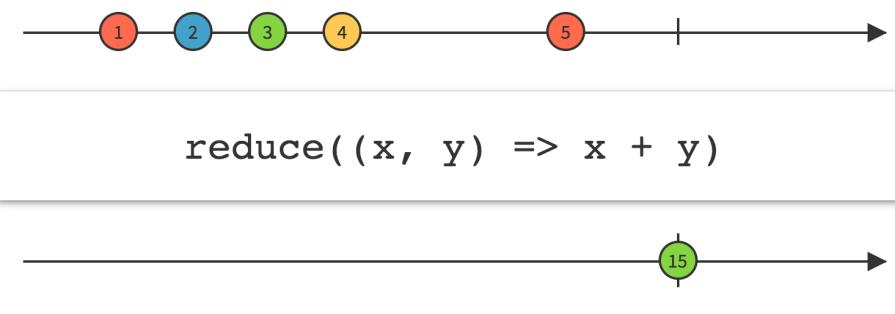
- Javadoc: [concat\(Observable\)](#)
- Javadoc: [concat\(Observable, Observable\)](#)

还有一个实例方法叫`concatwith`，这两者是等价的：

`Observable.concat(a, b)` 和 `a.concatwith(b)`。

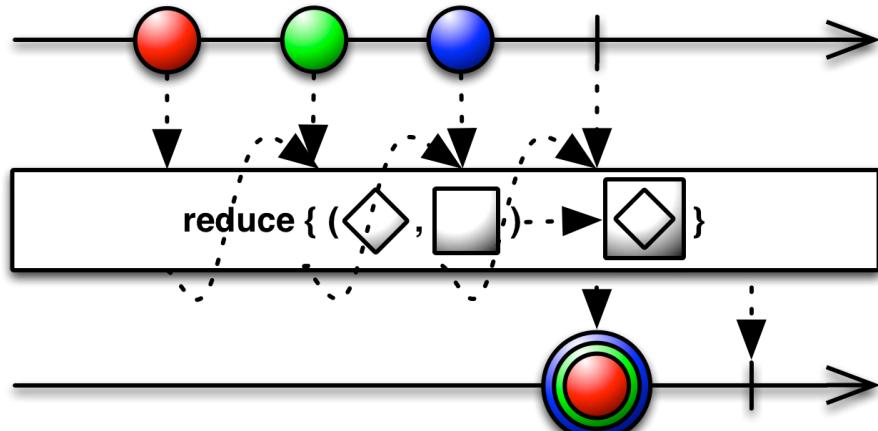
Reduce

按顺序对Observable发射的每项数据应用一个函数并发射最终的值



`Reduce`操作符对原始Observable发射数据的第一项应用一个函数，然后再将这个函数的返回值与第二项数据一起传递给函数，以此类推，持续这个过程知道原始Observable发射它的最后一项数据并终止，此时`Reduce`返回的Observable发射这个函数返回的最终值。

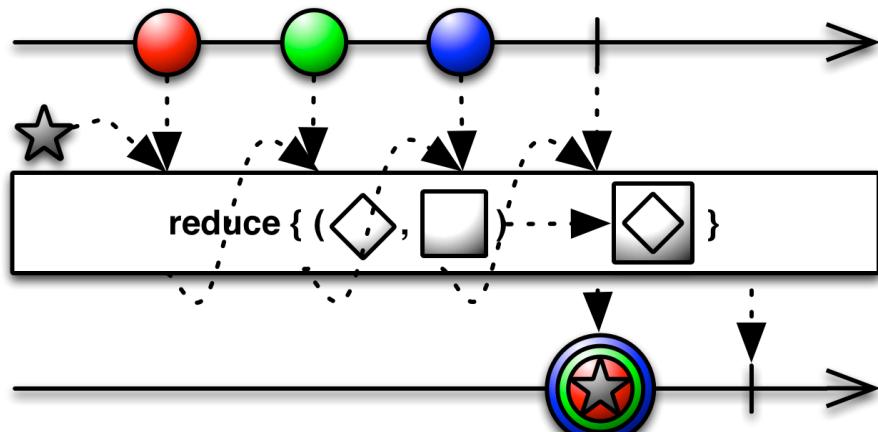
在其它场景中，这种操作有时被称为累积，聚集，压缩，折叠，注射等。



注意如果原始Observable没有发射任何数据，`reduce`抛出异常`IllegalArgumentException`。

`reduce`默认不在任何特定的调度器上执行。

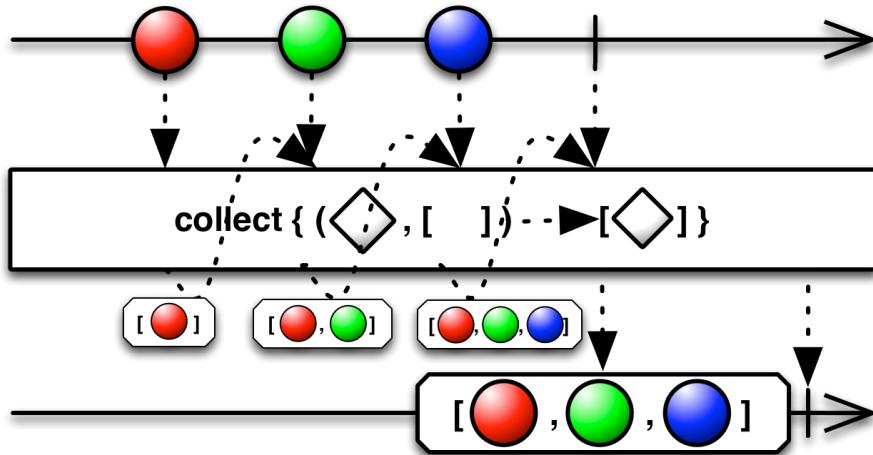
- Javadoc: `reduce(Func2)`



还有一个版本的`reduce`额外接受一个种子参数。注意传递一个值为`null`的种子是合法的，但是与不传种子参数的行为是不同的。如果你传递了种子参数，并且原始Observable没有发射任何数据，`reduce`操作符将发射这个种子值然后正常终止，而不是抛异常。

- Javadoc: `reduce(R,Func2)`

提示: 不建议使用`reduce`收集发射的数据到一个可变的数据结构, 那种场景你应该使用`collect`。



`collect`与`reduce`类似, 但它的目的是收集原始Observable发射的所有数据到一个可变的数据结构, `collect`生成的这个Observable会发射这项数据。它需要两个参数:

1. 一个函数返回可变数据结构
2. 另一个函数, 当传递给它这个数据结构和原始Observable发射的数据项时, 适当地修改数据结构。

`collect`默认不在任何特定的调度器上执行。

- Javadoc: `collect(Func0,Action2)`

异步操作

下面的这些操作符属于单独的`rxjava-async`模块, 它们用于将同步对象转换为Observable。

- **start()** – 创建一个Observable, 它发射一个函数的返回值
- **toAsync() or asyncAction() or asyncFunc()** – 将一个函数或者Action转换为已Observable, 它执行这个函数并发射函数的返回值
- **startFuture()** – 将一个返回Future的函数转换为一个Observable, 它发射Future的返回值
- **deferFuture()** – 将一个返回Observable的Future转换为一个Observable, 但是并不尝试获取这个Future返回的Observable, 直到有订阅者订阅它
- **forEachFuture()** – 传递Subscriber方法给一个Subscriber, 但是同时表现得像一个Future一样阻塞直到它完成
- **fromAction()** – 将一个Action转换为Observable, 当一个订阅者订阅时, 它执行这个action并发射它的返回值
- **fromCallable()** – 将一个Callable转换为Observable, 当一个订阅者订阅时, 它执行这个Callable并发射Callable的返回值, 或者发射异常
- **fromRunnable()** – convert a Runnable into an Observable that invokes the runnable and emits its result when a Subscriber

`subscribes`将一个Runnable转换为Observable，当一个订阅者订阅时，它执行这个Runnable并发射Runnable的返回值

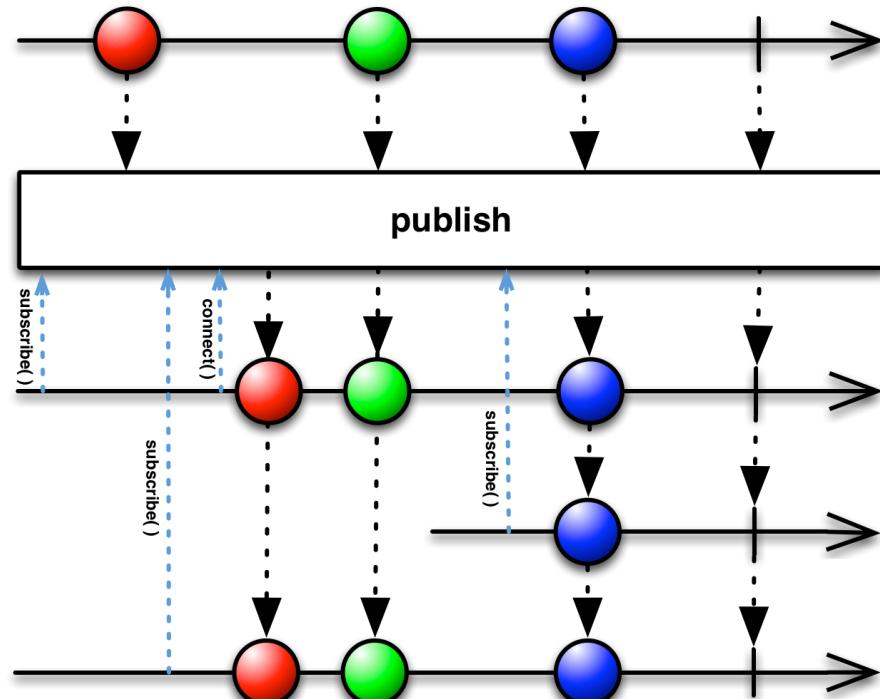
- `runAsync()` – 返回一个StoppableObservable，它发射某个Scheduler上指定的Action生成的多个actions

连接操作

这一节解释[ConnectableObservable](#) 和它的子类以及它们的操作符：

- `ConnectableObservable.connect()` – 指示一个可连接的Observable开始发射数据
- `Observable.publish()` – 将一个Observable转换为一个可连接的Observable
- `Observable.replay()` – 确保所有的订阅者看到相同的数据序列，即使它们在Observable开始发射数据之后才订阅
- `ConnectableObservable.refCount()` – 让一个可连接的Observable表现得像一个普通的Observable

一个可连接的Observable与普通的Observable差不多，除了这一点：可连接的Observable在被订阅时并不开始发射数据，只有在它的`connect()`被调用时才开始。用这种方法，你可以等所有的潜在订阅者都订阅了这个Observable之后才开始发射数据。



The following example code shows two Subscribers subscribing to the same Observable. In the first case, they subscribe to an ordinary Observable; in the second case, they subscribe to a Connectable Observable that only connects after both Subscribers subscribe. Note the difference in the output: 下面的示例代码展示了两个订阅者订阅同一个Observable的情况。第一种情形，它们订阅一个普通的Observable；第二种情形，它们订阅一个可连接的Observable，并且在两个都订阅后再连接。注意输出的不同：

示例 #1:

```
def firstMillion = Observable.range( 1, 1000000
).sample(7, java.util.concurrent.TimeUnit.MILLISECONDS);

firstMillion.subscribe(
    { println("Subscriber #1:" + it); },           // onNext
    { println("Error: " + it.getMessage()); }, // onError
    { println("Sequence #1 complete"); }           // onCompleted
);

firstMillion.subscribe(
    { println("Subscriber #2:" + it); },           // onNext
    { println("Error: " + it.getMessage()); }, // onError
    { println("Sequence #2 complete"); }           // onCompleted
);
Subscriber #1:211128
Subscriber #1:411633
Subscriber #1:629605
Subscriber #1:841903
Sequence #1 complete
Subscriber #2:244776
Subscriber #2:431416
Subscriber #2:621647
Subscriber #2:826996
Sequence #2 complete
```

示例 #2:

```
def firstMillion = Observable.range( 1, 1000000
).sample(7,
java.util.concurrent.TimeUnit.MILLISECONDS).publish();

firstMillion.subscribe(
    { println("Subscriber #1:" + it); },           // onNext
    { println("Error: " + it.getMessage()); }, // onError
    { println("Sequence #1 complete"); }           // onCompleted
);

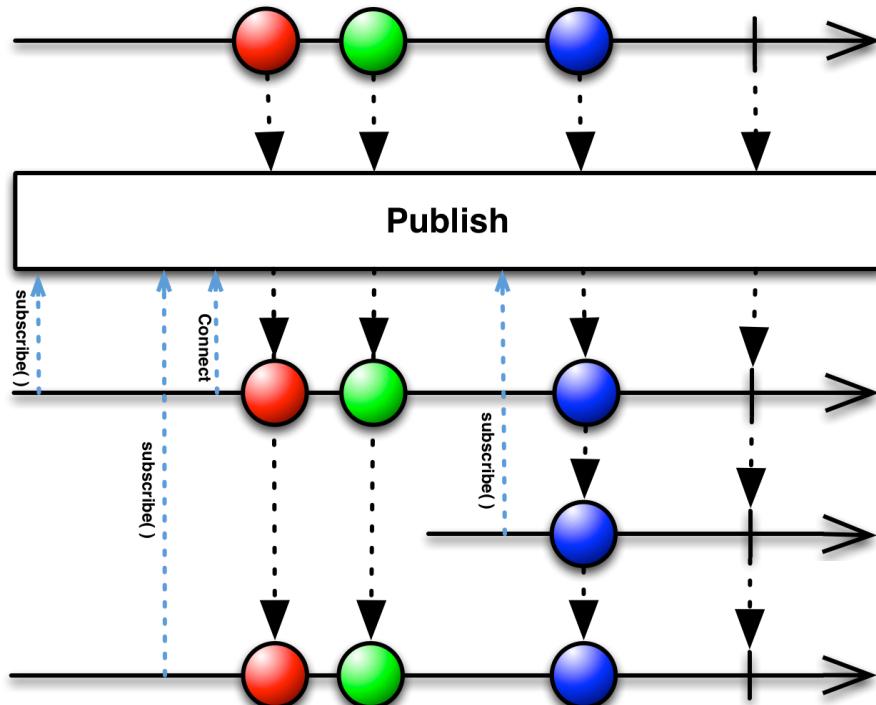
firstMillion.subscribe(
    { println("Subscriber #2:" + it); },           // onNext
    { println("Error: " + it.getMessage()); }, // onError
    { println("Sequence #2 complete"); }           // onCompleted
);

firstMillion.connect();
Subscriber #2:208683
```

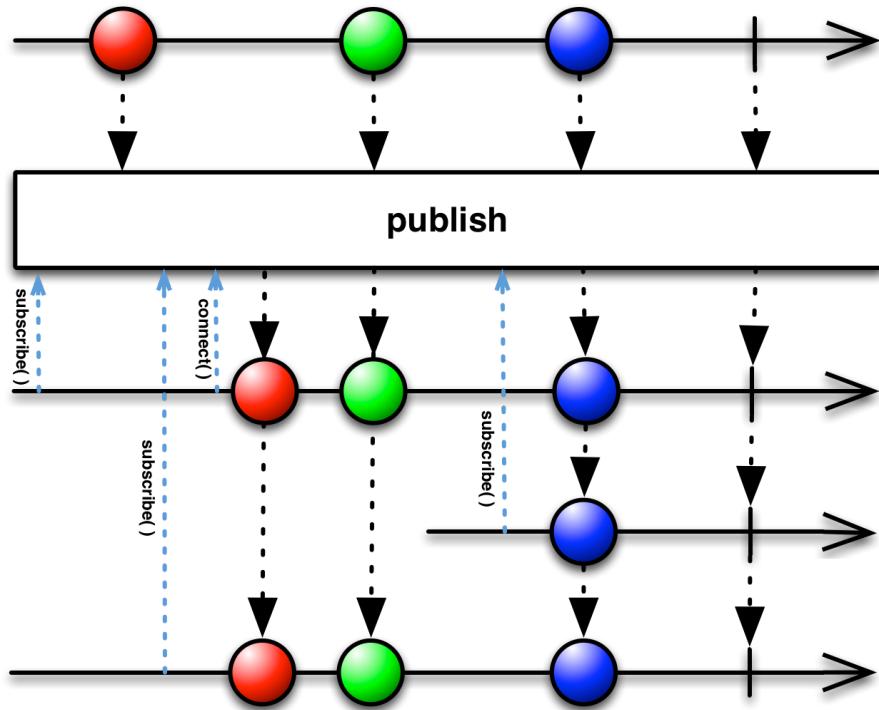
```
Subscriber #1:208683
Subscriber #2:432509
Subscriber #1:432509
Subscriber #2:644270
Subscriber #1:644270
Subscriber #2:887885
Subscriber #1:887885
Sequence #2 complete
Sequence #1 complete
```

Connect

让一个可连接的Observable开始发射数据给订阅者



可连接的Observable (*connectable Observable*)与普通的Observable差不多，不过它并不会在被订阅时开始发射数据，而是直到使用了 `Connect` 操作符时才会开始。用这个方法，你可以等待所有的观察者都订阅了Observable之后再开始发射数据。



RxJava中`connect`是`ConnectableObservable`接口的一个方法，使用`publish`操作符可以将一个普通的Observable转换为一个`ConnectableObservable`。

调用`ConnectableObservable`的`connect`方法会让它后面的Observable开始给发射数据给订阅者。

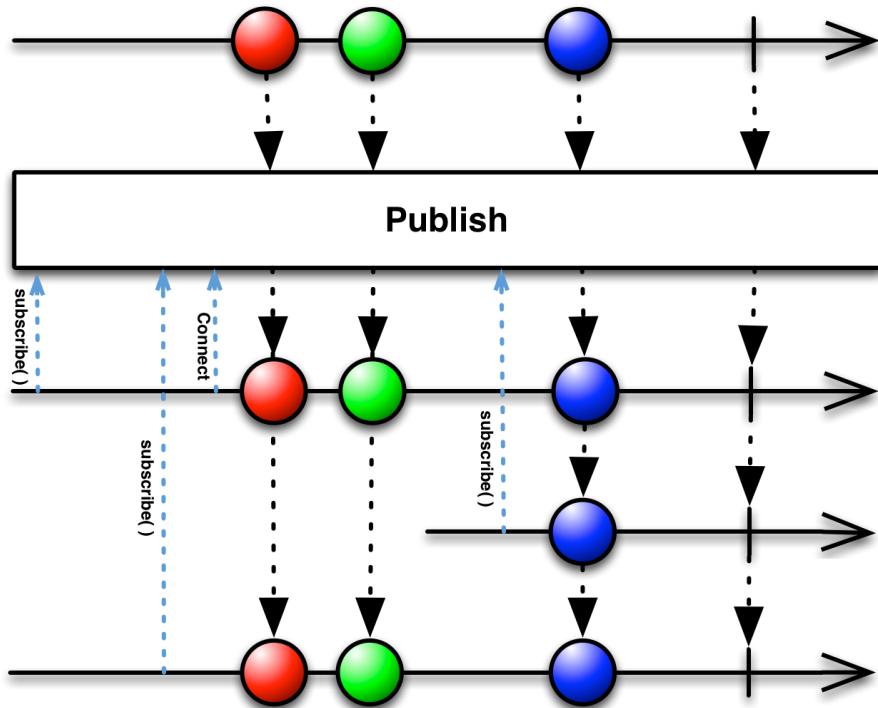
`connect`方法返回一个`Subscription`对象，可以调用它的`unsubscribe`方法让Observable停止发射数据给观察者。

即使没有任何订阅者订阅它，你也可以使用`connect`方法让一个Observable开始发射数据（或者开始生成待发射的数据）。这样，你可以将一个"冷"的Observable变为"热"的。

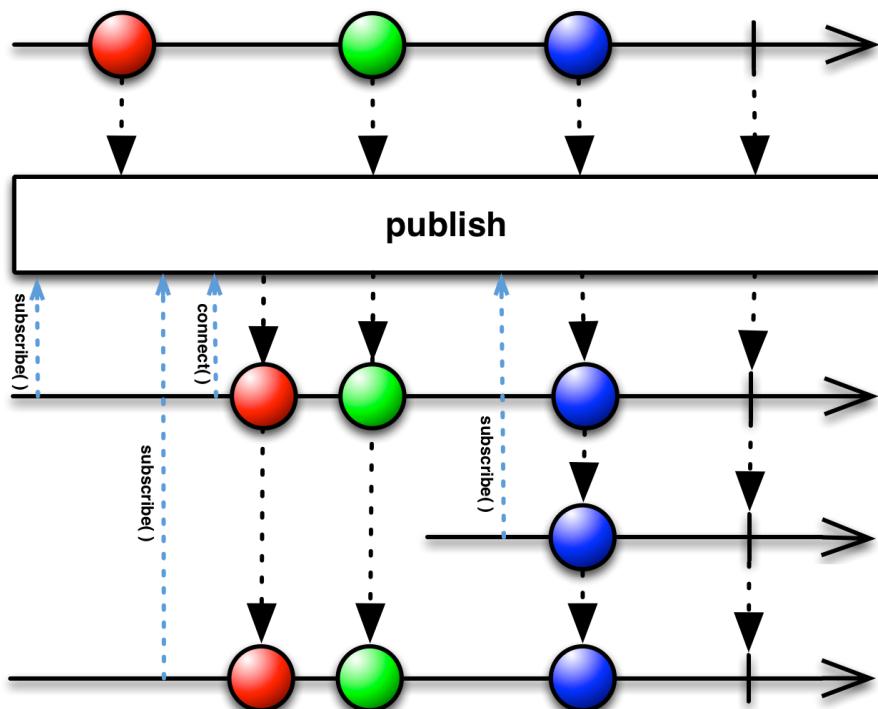
- Javadoc: `connect()`
- Javadoc: `connect(Action1)`

Publish

将普通的Observable转换为可连接的Observable

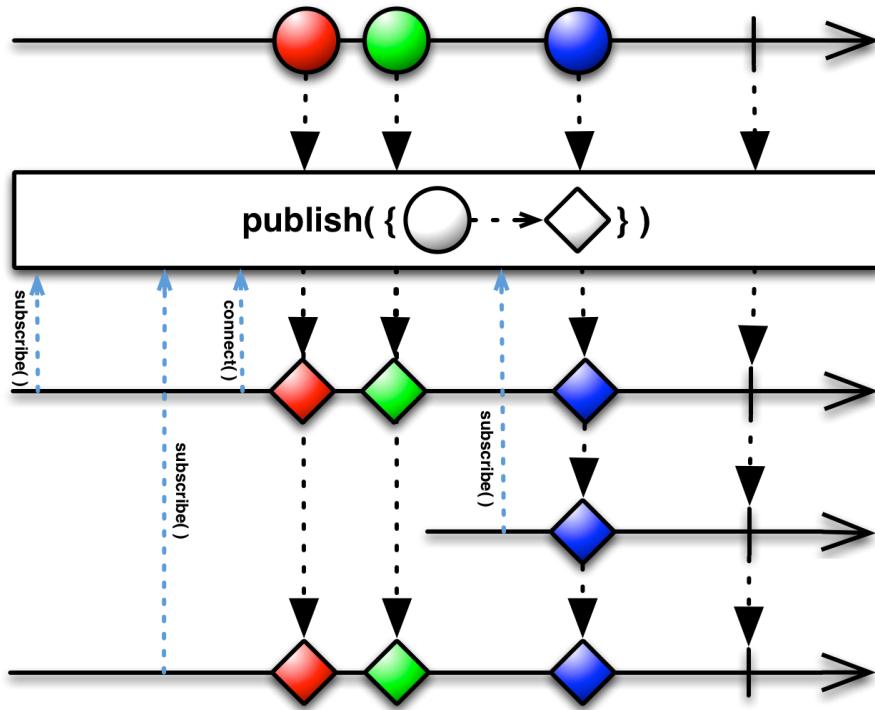


可连接的Observable (*connectable Observable*)与普通的Observable差不多，不过它并不会在被订阅时开始发射数据，而是直到使用了**Connect**操作符时才会开始。用这种方法，你可以在任何时候让一个Observable开始发射数据。



RxJava的实现为**publish**。

- Javadoc: [publish\(\)](#)

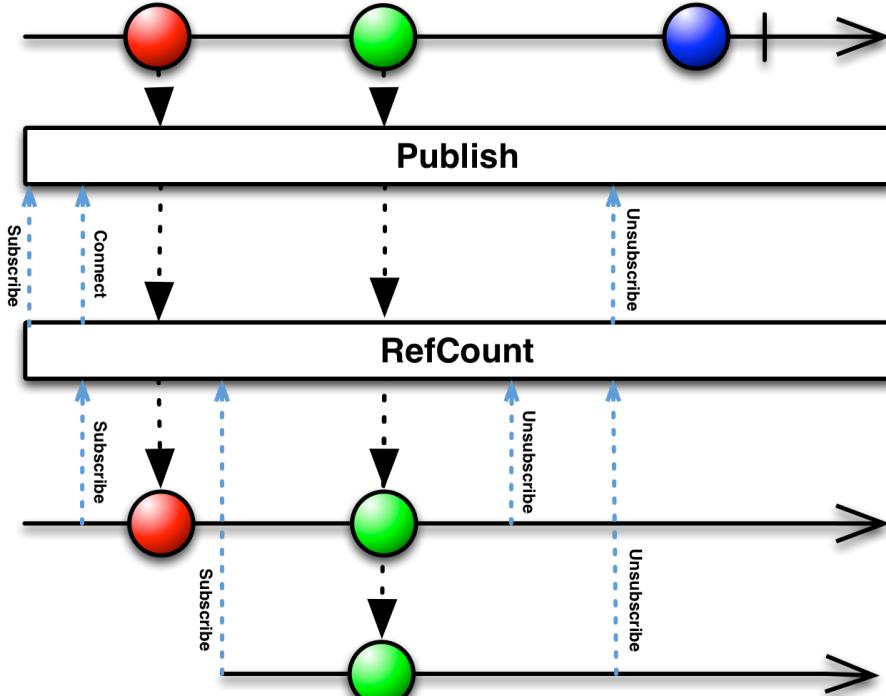


有一个变体接受一个函数作为参数。这个函数用原始Observable发射的数据作为参数，产生一个新的数据作为ConnectableObservable给发射，替换原位置的数据项。实质是在签名的基础上添加一个Map操作。

- Javadoc: [publish\(Func1\)](#)

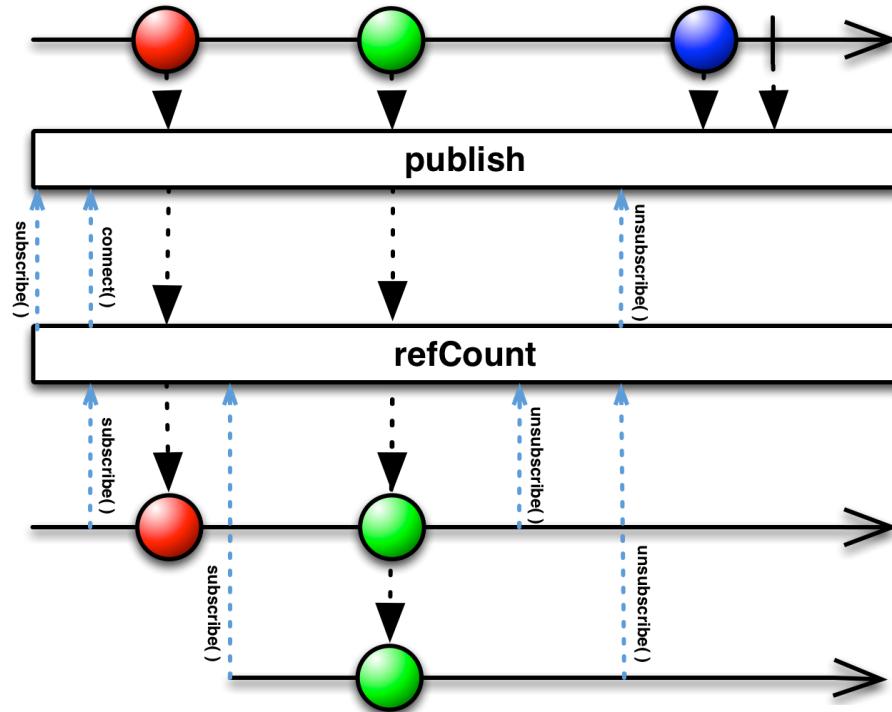
RefCount

让一个可连接的Observable行为像普通的Observable



可连接的Observable (*connectable Observable*)与普通的Observable差不多，不过它并不会在被订阅时开始发射数据，而是直到使用了 **Connect** 操作符时才会开始。用这种方法，你可以在任何时候让一个Observable开始发射数据。

`RefCount`操作符把从一个可连接的Observable连接和断开的过程自动化了。它操作一个可连接的Observable，返回一个普通的Observable。当第一个订阅者订阅这个Observable时，`RefCount`连接到下层的可连接Observable。`RefCount`跟踪有多少个观察者订阅它，直到最后一个观察者完成才断开与下层可连接Observable的连接。

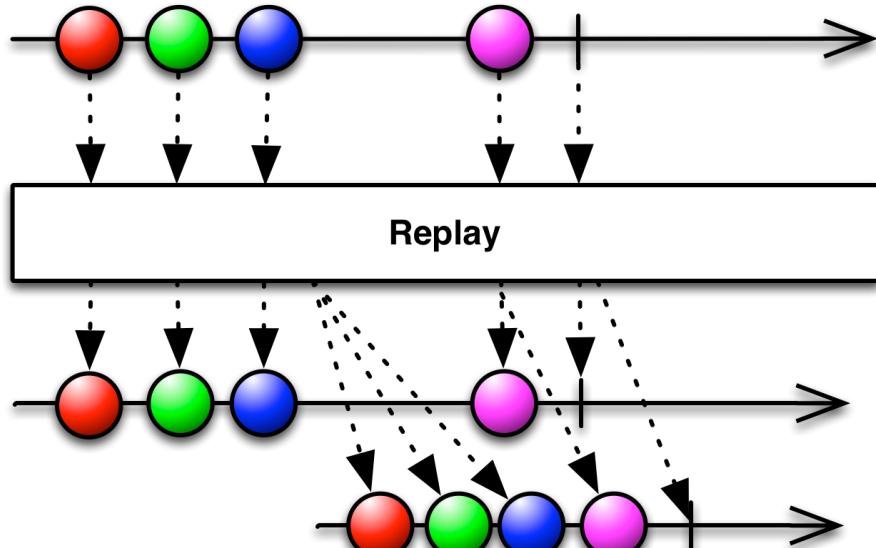


RxJava中的实现为 `refCount`，还有一个操作符叫 `share`，它的作用等价于对一个Observable同时应用 `publish` 和 `refCount` 操作。

- Javadoc: `RefCount()`
- Javadoc: `share()`

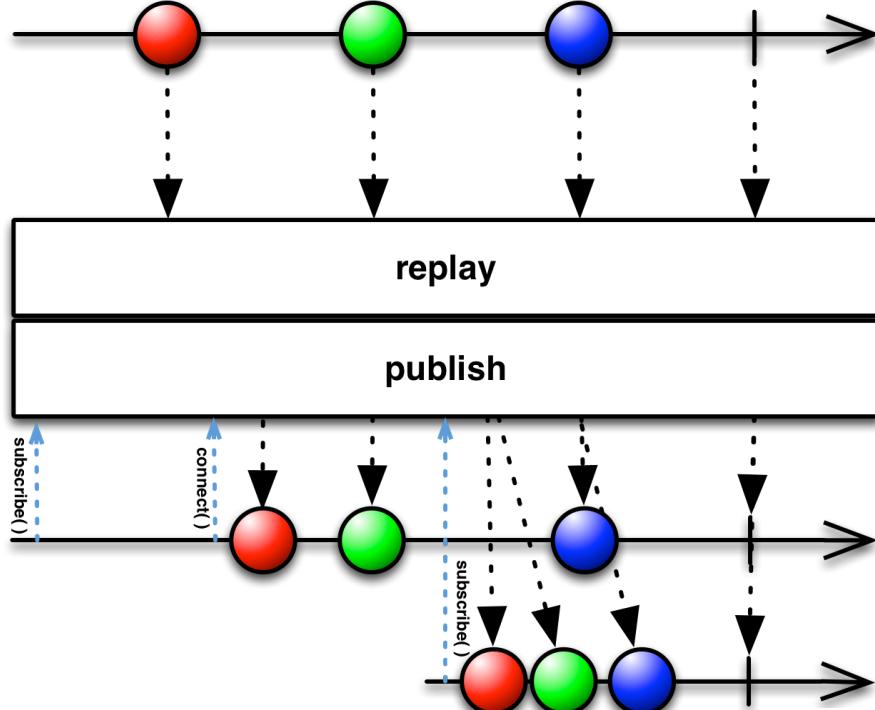
Replay

保证所有的观察者收到相同的数据序列，即使它们在Observable开始发射数据之后才订阅



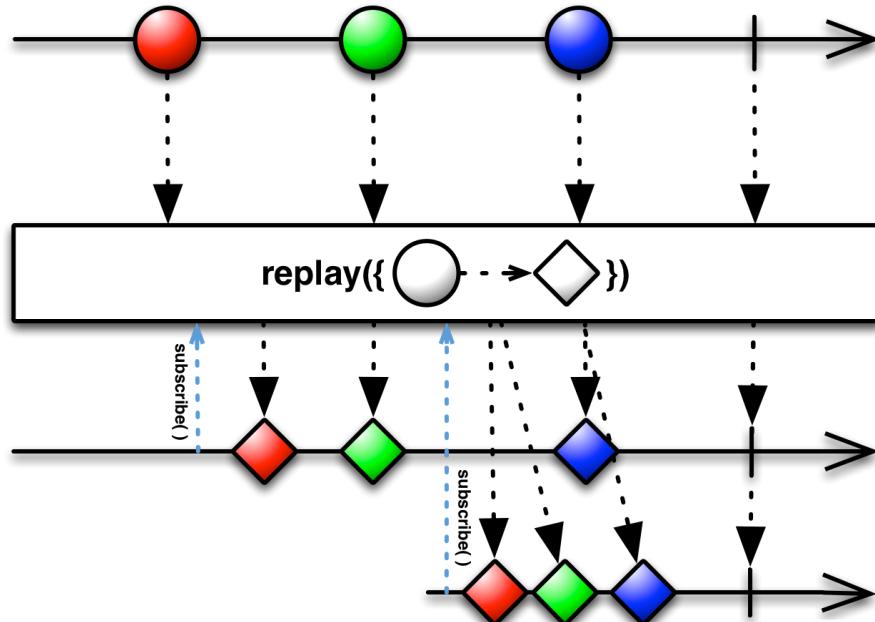
可连接的Observable (*connectable Observable*)与普通的Observable差不多，不过它并不会在被订阅时开始发射数据，而是直到使用了**Connect**操作符时才会开始。用这种方法，你可以在任何时候让一个Observable开始发射数据。

如果在将一个Observable转换为可连接的Observable之前对它使用**Replay**操作符，产生的这个可连接Observable将总是发射完整的数据序列给任何未来的观察者，即使那些观察者在这个Observable开始给其它观察者发射数据之后才订阅。



RxJava的实现为**replay**，它有多个接受不同参数的变体，有的可以指定**replay**的最大缓存数量，有的还可以指定调度器。

- Javadoc: [replay\(\)](#)
- Javadoc: [replay\(int\)](#)
- Javadoc: [replay\(long, TimeUnit\)](#)
- Javadoc: [replay\(int, long, TimeUnit\)](#)



有一种 `replay` 返回一个普通的Observable。它可以接受一个变换函数为参数，这个函数接受原始Observable发射的数据项为参数，返回结果Observable要发射的一项数据。因此，这个操作符其实是 `replay` 变换之后的数据项。

- Javadoc: `replay(Func1)`
- Javadoc: `replay(Func1,int)`
- Javadoc: `replay(Func1,long,TimeUnit)`
- Javadoc: `replay(Func1,int,long,TimeUnit)`

实现自己的操作符

你可以实现你自己的Observable操作符，本文展示怎么做。

如果你的操作符是被用于创造一个Observable，而不是变换或者响应一个Observable，使用 `create()` 方法，不要试图手动实现 `observable`。另外，你可以按照下面的用法说明创建一个自定义的操作符。

如果你的操作符是用于Observable发射的单独的数据项，按照下面的说明做：*Sequence Operators*。如果你的操作符是用于变换Observable发射的整个数据序列，按照这个说明做：*Transformational Operators*。

提示：在一个类似于Groovy的语言Xtend中，你可以以 *extension methods* 的方式实现你自己的操作符，不使用本文的方法，它们也可以链式调用。详情参见 [RxJava and Xtend](#)

序列操作符

下面的例子向你展示了怎样使用 `lift()` 操作符将你的自定义操作符（在这个例子中是 `myOperator`）与标准的RxJava操作符（如 `ofType` 和 `map`）一起使用：

```
fooObservable =
    barobservable ofType(Integer).map({it*2}).lift(new
        MyOperator<T>().map({"transformed by myOperator: " +
            it});
```

下面这部分向你展示了你的操作符的脚手架形式，以便它能正确的与 `lift()` 搭配使用。

实现你的操作符

将你的自定义操作符定义为实现了 `Operator` 接口的一个公开类，就像这样：

```
public class MyOperator<T> implements Operator<T> {
    public MyOperator( /* any necessary params here */ ) {
        /* 这里添加必要的初始化代码 */
    }
}
```

```

@Override
public Subscriber<? super T> call(Final Subscriber<? super T> s) {
    return new Subscriber<T>(s) {
        @Override
        public void onCompleted() {
            /* 这里添加你自己的onCompleted行为，或者仅仅传递完成通知： */
            if(!s.isUnsubscribed()) {
                s.onCompleted();
            }
        }

        @Override
        public void onError(Throwable t) {
            /* 这里添加你自己的onError行为，或者仅仅传递错误通知： */
            if(!s.isUnsubscribed()) {
                s.onError(t);
            }
        }

        @Override
        public void onNext(T item) {
            /* 这个例子对结果的每一项执行排序操作，然后返回这个结果 */
            if(!s.isUnsubscribed()) {
                transformedItem =
                    myOperatorTransformOperation(item);
                s.onNext(transformedItem);
            }
        }
    };
}

```

变换操作符

下面的例子向你展示了怎样使用 `compose()` 操作符将你得自定义操作符（在这个例子中，是一个名叫`myTransformer`的操作符，它将一个发射整数的 Observable转换为发射字符串的）与标准的RxJava操作符（如`ofType`和`map`）一起使用：

```

fooobservable =
    barobservable ofType(Integer).map({it*2}).compose(new
    MyTransformer<Integer, String>().map({"transformed by
    myoperator: " + it});

```

下面这部分向你展示了你的操作符的脚手架形式，以便它能正确的与 `compose()` 搭配使用。

实现你的变换器

将你的自定义操作符定义为实现了 `Transformer` 接口的一个公开类，就像这样：

```
public class MyTransformer<Integer, String> implements
Transformer<Integer, String> {
    public MyTransformer( /* any necessary params here */ )
    {
        /* 这里添加必要的初始化代码 */
    }

    @Override
    public Observable<String> call(Observable<Integer>
source) {
        /*
         * 这个简单的例子Transformer应用一个map操作,
         * 这个map操作将发射整数变换为发射整数的字符串表示。
         */
        return source.map( new Func1<Integer, String>() {
            @Override
            public String call(Integer t1) {
                return String.valueOf(t1);
            }
        });
    }
}
```

参见

- “Don’t break the chain: use RxJava’s `compose()` operator” by Dan Lew

其它需要考虑的

-
- 在发射任何数据（或者通知）给订阅者之前，你的序列操作符可能需要检查它的 `Subscriber.isUnsubscribed()` 状态，如果没有订阅者了，没必要浪费时间生成数据项。
 - 请注意：你的序列操作符必须复合Observable协议的核心原则：
 - 它可能调用订阅者的 `onNext()` 方法任意次，但是这些调用必须是不重叠的。
 - 它只能调用订阅者的 `onCompleted()` 或 `onError()` 正好一次，但不能都调用，而且不能在这之后调用订阅者的 `onNext()` 方法。

- 如果你不能保证你得操作符遵从这两个原则，你可以给它添加 `serialize()` 操作符，它会强制保持正确的行为。
- 请关注这里 [Issue #1962](#) —— 需要有一个计划创建一个测试脚手架，你可以用它来写测试验证你的新操作符遵从了 Observable 协议。
- 不要让你的操作符阻塞别的操作。
- When possible, you should compose new operators by combining existing operators, rather than implementing them with new code. RxJava itself does this with some of its standard operators, for example:
- 如果可能，你应该组合现有的操作符创建你的新操作符，而不是从零开始实现它。RxJava自身的标准操作符也是这样做的，例如：
 - `first()` 被定义为 `take(1).single()`
 - `ignoreElements()` 被定义为 `filter(alwaysFalse())`
 - `reduce(a)` 被定义为 `scan(a).last()`
- 如果你的操作符使用了函数或者lambda表达式作为参数，请注意它们可能是异常的来源，而且要准备好捕获这些异常，并且使用

onError()

通知订阅者。

- 某些异常被认为是致命的，对它们来说，调用 `onError()` 毫无意义，那样或者是无用的，或者只是对问题的妥协。你可以使用 `Exceptions.throwIfFatal(throwable)` 方法过滤掉这些致命的异常，并重新抛出它们，而不是试图发射关于它们的通知。
- 一般说来，一旦发生错误应立即通知订阅者，而不是首先尝试发射更多的数据。
- 请注意 `null` 可能是 Observable 发射的一个合法数据。频繁发生错误的一个来源是：测试一些变量并且将持有一个非 `null` 值作为是否发射了数据的替代。一个值为 `null` 的数据仍然是一个发射数据项，它与没有发射任何东西是不能等同的。
- 想让你的操作符在反压(*backpressure*)场景中变得得好可能会非常棘手。可以参考 Dávid Karnok 的博客 [Advanced RxJava](#)，这里有一个涉及到的各种因素和怎样处理它们的很值得看的讨论。

插件让你可以用多种方式修改 RxJava 的默认行为：

- 修改默认的计算、IO 和新线程调度器集合
- 为 RxJava 可能遇到的特殊错误注册一个错误处理器
- 注册一个函数记录一些常规 RxJava 活动的发生

RxJavaSchedulersHook

这个插件让你可以使用你选择的调度器覆盖默认的计算、IO 和新线程调度 (`Scheduler`)，要做到这些，需要继承 `RxJavaSchedulersHook` 类并覆写这些方法：

- `Scheduler getComputationscheduler()`

- `scheduler getIOScheduler()`
- `scheduler getNewThreadScheduler()`
- `Action0 onSchedule(action)`

然后是下面这些步骤：

1. 创建一个你实现的 `RxJavaSchedulersHook` 子类的对象。
2. 使用 `RxJavaPlugins.getInstance()` 获取全局的 RxJavaPlugins 对象。
3. 将你的默认调度器对象传递给 `RxJavaPlugins` 的 `registerSchedulersHook()` 方法。

完成这些后，RxJava 会开始使用你的方法返回的调度器，而不是内置的默认调度器。

RxJavaErrorHandler

这个插件让你可以注册一个函数处理传递给 `Subscriber.onError(Throwable)` 的错误。要做到这一点，需要继承 `RxJavaErrorHandler` 类并覆写这个方法：

- `void handleError(Throwable e)`

然后是下面这些步骤：

1. 创建一个你实现的 `RxJavaErrorHandler` 子类的对象。
2. 使用 `RxJavaPlugins.getInstance()` 获取全局的 RxJavaPlugins 对象。
3. 将你的错误处理器对象传递给 `RxJavaPlugins` 的 `registerErrorHandler()` 方法。

完成这些后，RxJava 会开始使用你的错误处理器处理传递给 `Subscriber.onError(Throwable)` 的错误。

RxJavaObservableExecutionHook

这个插件让你可以注册一个函数用于记录日志或者性能数据收集，RxJava 在某些常规活动时会调用它。要做到这一点，需要继承 `RxJavaObservableExecutionHook` 类并覆写这些方法：

方法	何时调用
<code>onCreate()</code>	在 <code>observable.create()</code> 方法中
<code>onSubscribeStart()</code>	在 <code>observable.subscribe()</code> 之前立刻
<code>onSubscribeReturn()</code>	在 <code>observable.subscribe()</code> 之后立刻
<code>onSubscribeError()</code>	在 <code>observable.subscribe()</code> 执行失败时
<code>onLift()</code>	在 <code>observable.lift()</code> 方法中

然后是下面这些步骤：

1. 创建一个你实现的 `RxJavaObservableExecutionHook` 子类的对象。
2. 使用 `RxJavaPlugins.getInstance()` 获取全局的RxJavaPlugins对象。
3. 将你的Hook对象传递给 `RxJavaPlugins` 的 `registerObservableExecutionHook()` 方法。

When you do this, RxJava will begin to call your functions when it encounters the specific conditions they were designed to take note of. 完成这些后，在满足某些特殊的条件时，RxJava会开始调用你的方法。

背压问题

背压是指在异步场景中，被观察者发送事件速度远快于观察者的处理速度的情况下，一种告诉上游的被观察者降低发送速度的策略

简而言之，背压是流速控制的一种策略。

需要强调两点：

- 背压策略的一个前提是异步环境，也就是说，被观察者和观察者处在不同的线程环境中。
- 背压（Backpressure）并不是一个像flatMap一样可以在程序中直接使用的操作符，他只是一种控制事件流速的策略。

响应式拉取（reactive pull）

首先我们回忆之前那篇《关于Rxjava最友好的文章》，里面其实提到，在RxJava的观察者模型中，被观察者是主动的推送数据给观察者，观察者是被动接收的。而响应式拉取则反过来，观察者主动从被观察者那里去拉取数据，而被观察者变成被动的等待通知再发送数据。

结构示意图如下：

观察者可以根据自身实际情况按需拉取数据，而不是被动接收（也就相当于告诉上游观察者把速度慢下来），最终实现了上游被观察者发送事件的速度的控制，实现了背压的策略。

源码

```
public class FlowableOnBackpressureBufferStrategy{  
    ...  
    @Override  
    public void onNext(T t) {
```

```
        if (done) {
            return;
        }
        boolean callOnOverflow = false;
        boolean callError = false;
        Deque<T> dq = deque;
        synchronized (dq) {
            if (dq.size() == bufferSize) {
                switch (strategy) {
                    case DROP_LATEST:
                        dq.pollLast();
                        dq.offer(t);
                        callOnOverflow = true;
                        break;
                    case DROP_OLDEST:
                        dq.poll();
                        dq.offer(t);
                        callOnOverflow = true;
                        break;
                    default:
                        // signal error
                        callError = true;
                        break;
                }
            } else {
                dq.offer(t);
            }
        }

        if (callOnOverflow) {
            if (onOverflow != null) {
                try {
                    onOverflow.run();
                } catch (Throwable ex) {
                    Exceptions.throwIfFatal(ex);
                    s.cancel();
                    onError(ex);
                }
            }
        } else if (callError) {
            s.cancel();
            onError(new
MissingBackpressureException());
        } else {
            drain();
        }
    }
    ...
}
```

在这段源码中，根据不同的背压策略进行了不同的处理措施，当然这只是列举了一段关于buffer背压策略的例子。

根源

产生背压问题的根源就是上游发送速度与下游的处理速度不均导致的，所以如果想要解决这个问题就需要通过匹配两个速率达到解决这个背压根源的措施。

通常有两个策略可供使用：

1. 从数量上解决，对数据进行采样
2. 从速度上解决，降低发送事件的速率
3. 利用flowable和subscriber

使用Flowable

```
Flowable<Integer> upstream = Flowable.create(new
    FlowableOnSubscribe<Integer>() {
        @Override
        public void subscribe(FlowableEmitter<Integer>
            emitter) throws Exception {
            Log.d(TAG, "emit 1");
            emitter.onNext(1);
            Log.d(TAG, "emit 2");
            emitter.onNext(2);
            Log.d(TAG, "emit 3");
            emitter.onNext(3);
            Log.d(TAG, "emit complete");
            emitter.onComplete();
        }
    }, BackpressureStrategy.ERROR); //增加了一个参数

    Subscriber<Integer> downstream = new
    Subscriber<Integer>() {

        @Override
        public void onSubscribe(Subscription s) {
            Log.d(TAG, "onSubscribe");
            s.request(Long.MAX_VALUE); //注意这句代码
        }

        @Override
        public void onNext(Integer integer) {
            Log.d(TAG, "onNext: " + integer);

        }
    }
}
```

```
    @Override
    public void onError(Throwable t) {
        Log.w(TAG, "onError: ", t);
    }

    @Override
    public void onComplete() {
        Log.d(TAG, "onComplete");
    }
};

upstream.subscribe(downstream);
```

我们注意到这次和 `Observable` 有些不同。首先是创建 `Flowable` 的时候增加了一个参数，这个参数是用来选择背压，也就是出现上下游流速不均衡的时候应该怎么处理的办法，这里我们直接用 `BackpressureStrategy.ERROR` 这种方式，这种方式会在出现上下游流速不均衡的时候直接抛出一个异常，这个异常就是著名的 `MissingBackpressureException`。其余的策略后面再来讲解。

另外的一个区别是在下游的 `onSubscribe` 方法中传给我们的不再是 `Disposable` 了，而是 `Subscription`，它俩有什么区别呢，首先它们都是上下游中间的一个开关，之前我们说调用 `Disposable.dispose()` 方法可以切断水管，同样的调用 `Subscription.cancel()` 也可以切断水管，不同的地方在于 `Subscription` 增加了一个 `void request(long n)` 方法，这个方法有什么用呢，在上面的代码中也有这么一句代码：

```
s.request(Long.MAX_VALUE);
```

这是因为 `Flowable` 在设计的时候采用了一种新的思路也就是 `响应式拉取` 的方式来更好的解决上下游流速不均衡的问题，与我们之前所讲的控制数量和控制速度不太一样，这种方式用通俗易懂的话来说就好比是 `叶问打鬼子`，我们把 `上游` 看成 `小日本`，把 `下游` 当作 `叶问`，当调用 `Subscription.request(1)` 时，`叶问` 就说我要打一个！然后 `小日本` 就拿出一个鬼子给 `叶问`，让他打，等 `叶问` 打死这个鬼子之后，再次调用 `request(10)`，`叶问` 就又说我要打十个！然后 `小日本` 又派出十个鬼子给 `叶问`，然后就在边上热闹，看 `叶问` 能不能打死十个鬼子，等 `叶问` 打死十个鬼子后再继续要鬼子接着打...

所以我们要把 `request` 当做是一种能力，当成下游处理事件的能力，下游能处理几个就告诉上游我要几个，这样只要上游根据下游的处理能力来决定发送多少事件，就不会造成一窝蜂的发出一堆事件来，从而导致 OOM。这也就完美的解决之前我们所学到的两种方式的缺陷，过滤事件会导致事件丢失，减速又可能导致性能损失。而这种方式既解决了事件丢失的问题，又解决了速度的问题，完美！

同步情况

```
Observable.create(new ObservableOnSubscribe<Integer>() {
```

```

@Override

    public void subscribe(ObservableEmitter<Integer>
emitter) throws Exception {
        for (int i = 0; ; i++) { //无限循环发事件

            emitter.onNext(i);

        }
    }

}).subscribe(new Consumer<Integer>() {

    @Override

    public void accept(Integer integer) throws Exception {

        Thread.sleep(2000);

        Log.d(TAG, "" + integer);

    }
});

}

```

当上下游工作在同一个线程中时，这时候是一个同步的订阅关系，也就是说上游每发送一个事件必须等到下游接收处理完了以后才能接着发送下一个事件。

同步与异步的区别就在于有没有缓存发送事件的缓冲区。

异步情况

通过subscribeOn和observeOn来确定对应的线程，达到异步的效果，异步时会有一个对应的缓存区来换从上游发送的事件。

```

public enum BackpressureStrategy {

    /**
     * OnNext events are written without any buffering or
     * dropping.
     *
     * Downstream has to deal with any overflow.
     *
     * <p>Useful when one applies one of the custom-
     * parameter onBackpressureXXX operators.
     */
    MISSING,
    /**
     * Signals a MissingBackpressureException in case the
     * downstream can't keep up.
     */
}

```

```

        ERROR,
        /**
         * Buffers <em>all</em> onNext values until the
downstream consumes it.
        */
        BUFFER,
        /**
         * Drops the most recent onNext value if the
downstream can't keep up.
        */
        DROP,
        /**
         * Keeps only the latest onNext value, overwriting any
previous value if the
         * downstream can't keep up.
        */
        LATEST
    }
}

```

背压策略:

1. error, 缓冲区大概在128
2. buffer, 缓冲区在1000左右
3. drop, 把存不下的事件丢弃
4. latest, 只保留最新的
5. missing, 缺省设置, 不做任何操作

上游从哪里得知下游的处理能力呢？我们来看看上游最重要的部分，肯定就是**FlowableEmitter**了啊，我们就是通过它来发送事件的啊，来看看它的源码吧（别紧张，它的代码灰常简单）：

```

public interface FlowableEmitter<T> extends Emitter<T> {
    void setDisposable(Disposable s);
    void setCancellable(Cancellable c);

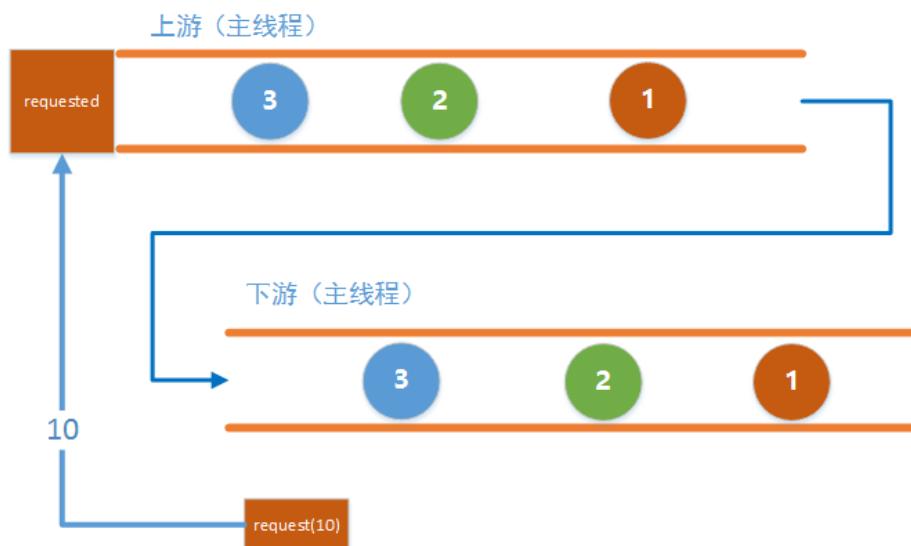
    /**
     * The current outstanding request amount.
     * <p>This method is thread-safe.
     * @return the current outstanding request amount
    */
    long requested();

    boolean isCancelled();
    FlowableEmitter<T> serialize();
}

```

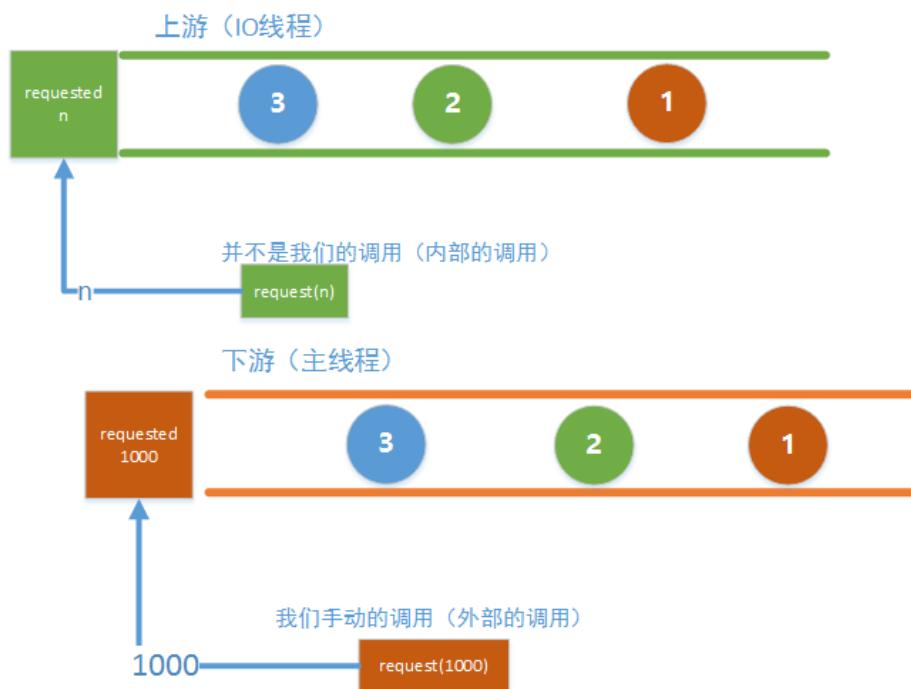
FlowableEmitter是个接口，继承Emitter，Emitter里面就是我们的onNext(), onComplete()和onError()三个方法。我们看到FlowableEmitter中有这么一个方法：

```
long requested();
```



同步request.png

这张图的意思就是当上下游在同一个线程中的时候，在下游调用`request(n)`就会直接改变上游中的`requested`的值，多次调用便会叠加这个值，而上游每发送一个事件之后便会去减少这个值，当这个值减少至0的时候，继续发送事件便会抛异常了。



异步request.png

可以看到，当上下游工作在不同的线程里时，每一个线程里都有一个`requested`，而我们调用`request(1000)`时，实际上改变的是下游主线程中的`requested`，而上游中的`requested`的值是由RxJava内部调用`request(n)`去设置的，这个调用会在合适的时候自动触发。

Rxjava实例开发应用

1. 网络请求处理(轮询，嵌套，出错重连)
2. 功能防抖
3. 从多级缓存获取数据
4. 合并数据源
5. 联合判断
6. 与 Retrofit,RxBinding,EventBus结合使用

Rxjava原理

1. Scheduler线程切换工作原理
2. 数据的发送与接收(观察者模式)
3. lift的工作原理
4. map的工作原理
5. flatMap的工作原理
6. merge的工作原理
7. concat的工作原理